OSS-Fuzzgen: Automated Fuzzing of Open Source Java Projects

Sheung Chi Chan
Ada Logics
London, England, UK
arthur.chan@adalogics.com

Adam Korczynski
Ada Logics
London, England, UK
adam@adalogics.com

David Korczynski
Ada Logics
London, England, UK
david@adalogics.com

Abstract*—OSS-Fuzz is an open source service for managing the fuzzing of open source projects. Open source projects integrate into OSS-Fuzz by adding a set of fuzzing harnesses targeting their project and relevant build logic for the OSS-Fuzz infrastructure. OSS-Fuzz will then build and run these harnesses continuously and report when finding any security or reliability issues. To date, OSS-Fuzz has reported tens of thousands of bugs in software and the list is continuously growing. Unfortunately, the process of integrating projects into OSS-Fuzz is still largely manual and both the creation of fuzzing harnesses and build setup are time-consuming tasks. In this paper, we propose OSS-Fuzzgen, a system that can automatically generate OSS-Fuzz integrations for open source Java projects, including fuzzing harness synthesis and build infrastructure generation. The input to OSS-Fuzzgen is a GitHub URL to a given open source project. The output is a list of ranked OSS-Fuzz integration candidates that can be run by OSS-Fuzz. We empirically evaluate our setup by running the system through more than 200 open source projects, which resulted in more than 100 generated OSS-Fuzz integrations. We manually inspect the results and submit 31 of these to OSS-Fuzz resulting in more than 50 reported bugs across the 31 projects. For 11 of these bugs, we submitted fixes to the relevant open source projects, and 9 fixes were accepted and merged into the upstream open source project. We have open-sourced OSS-Fuzzgen and the code is available on GitHub[1].

Keywords*—*OSS-Fuzz; Fuzz-Introspector; Java; fuzzing; security testing; libfuzzer.

I. Introduction

Fuzzing is an effective technique for finding security and reliability issues in software. The high-level idea behind fuzzing is to pass arbitrary inputs to a given application and monitor if unexpected behaviour happens. There are many success stories from fuzzing, both in terms of finding difficult-to-catch issues and also rapidly catching regressions in software. OSS-Fuzz is an open source fuzzing service that currently manages fuzzing infrastructure for more than 1000 widely used open source projects and has reported tens of thousands of security and reliability bugs in these projects[2][3].

To integrate fuzzing in a project, coverage-feedback fuzzing specifically, the general approach is to write fuzzing harnesses that execute the target software package with input seeded by data from the fuzzing engine. In coverage-guided fuzzing, a harness is a small program that aims to explore the target code base by continuously mutating its input and collecting seeds (inputs) that trigger unique code execution in the target. They are often similar to unit tests, but instead of testing a specific input, the harnesses test a generalised domain of inputs, and the domain is often much larger than what is feasible to brute-force, e.g. arbitrarily large buffer, hence the use of genetic mutational algorithms in the fuzzing engine. During execution, the fuzzing engine observes the execution of the target code and uses coverage data to guide the input generation and mutation, resulting in the creation of inputs to the fuzzing harness that incrementally explore the target code base[4].

The harnesses comprise a central role when fuzzing a software package and many projects have several harnesses to trigger different parts of the project's code base. For example, OSS-Fuzz has around 1100 projects integrated into the fuzzing service but runs more than 4500 fuzzing harnesses daily[5]. Another central component when fuzzing is to have a build infrastructure in place that makes it possible to build the target software using an environment that supports fuzzing. Specifically, the target codebase needs to be instrumented appropriately, which happens during the compilation stage and the harnesses need to be appropriately linked to the project.

The process of writing harnesses for a software package as well as constructing the built environment that makes fuzzing possible is cumbersome and time-consuming. It can often take several weeks to integrate fuzzing into medium-sized software packages, and many years to integrate fuzzing into extensive code bases such as modern browsers or operating systems. Furthermore, despite the success of OSS-Fuzz fuzzing more than 1100 software packages continuously, there remain tens of thousands of open source software packages that are not being fuzzed.

There has been efforts into automating fuzzing harness writing[6][7][8][9][10][11] and also related efforts for inferring API specifications[12][13]. In general, a fuzzing harness requires the effort from OSS-Fuzz to observe and mutate the input to extensively cover the underlying code base of the target projects. Otherwise, there is no difference compared to unit testing. These efforts are, however, not targeted open source projects and are only generating fuzzing harnesses but not the full OSS-Fuzz integrations. Some of the efforts require manual studying of the target projects to specify target methods or classes for the fuzzing harnesses generation. This setting makes it difficult to automatically generate the full OSS-Fuzz integration and requires extensive manual efforts before and after the automatic generation process.

In this paper, we introduce OSS-Fuzzgen, a system for automatically generating OSS-Fuzz integrations for open source Java projects. Our system takes as input a list of GitHub repositories and will output a set of fuzzing harnesses and build infrastructure for the projects such that the projects can be fuzzed by way of OSS-Fuzz. Our system relies on static and dynamic program analysis techniques, which are developed as extensions to Fuzz Introspector[14]. To verify our system, we present an empirical evaluation of running our system against 257 open source projects which resulted in more than 100 possible project submissions to OSS-Fuzz. We submit 31 resulting projects with high coverage and fuzzing performance to OSS-Fuzz. For several of the bugs found by the generated harnesses, we reported them to the relevant open source projects, which confirmed that the bugs found were legit and accepted our patches to fix the issues.

Contributions This paper makes the following contributions:

- We present a novel system for automatically synthesising Java fuzzing harnesses.
- We present the first system to automatically construct OSS-Fuzz project integrations.
- We present an extensive empirical evaluation of more than 200 open source projects and verify that our system finds real bugs in widely used Java projects.

The remainder of this paper is structured as follows. In Section II, we introduce the OSS-Fuzz and Fuzz Introspector services. In Section III, we give an overview of the OSS-Fuzzgen tool. In Section IV, we illustrate the detailed design of the OSS-Fuzzgen tool. In Section V, we give details of the empirical evaluation of the OSS-Fuzzgen tool. We then discuss the limitation and future enhancement plan for the OSS-Fuzzgen tool in Section VI and conclude the paper in Section VII.

II. Background

In this section, we introduce OSS-Fuzz and Fuzz Introspector, each comprising a central role in our system. Specifically, our solution is built as an extension to Fuzz Introspector while we rely on OSS-Fuzz as the runtime environment for our generated harnesses.

A. OSS-Fuzz

OSS-Fuzz[2] is a free online service that manages the execution of fuzzing harnesses for open source projects. The process for integrating into the service is that an open source project develops a set of fuzzing harnesses targeting the project and also some necessary glue for OSS-Fuzz to build these harnesses. This glue is composed of a *project.yaml* file with metadata, a *Dockerfile* to construct the container in which the harnesses are built and also a shell script, *build.sh*, that holds the commands for building the target project and harnesses inside the container.

To submit the project for OSS-Fuzz integration, a pull request is made to the OSS-Fuzz repository with the specific glue in the dedicated project directory. Once the pull request is merged OSS-Fuzz will daily build the fuzzing harnesses using the latest upstream code. OSS-Fuzz then runs these harnesses for an extended period and reports to the people listed in the *project.yaml* metadata if any of the harnesses find any bugs. OSS-Fuzz provides the infrastructure to build and run harnesses locally for each project integrated into OSS-Fuzz. In this way, there is a unified interface for building and running more than 4500 fuzzing harnesses spread across more than 1100 projects.

B. Fuzz Introspector

Fuzz Introspector[14] is a tool for providing introspection capabilities into the fuzzing of a given software package. Fuzz Introspector can, for example, analyse the static reachability of fuzzing harnesses, find candidate methods in the target code that are likely good targets for fuzzing and combine runtime code coverage data with static analysis capabilities to identify potential runtime blockers for the fuzzing harnesses [15].

Fuzz Introspector is architecturally split between multiple frontends and a single backend. The frontends are language-specific static analysis tools, often in the form of compiler extensions, which extract data about the software under analysis. The Java frontend of Fuzz Introspector is built on top of SOOT[16] and this is the primary component of Fuzz Introspector that OSS-Fuzzgen uses.

III. OSS-Fuzzgen Overview

OSS-Fuzzgen takes as input one or more URLs to a given set of open source projects on GitHub. OSS-Fuzzgen outputs a set of OSS-Fuzz integrations for each of the provided open source projects, where each integration includes the base OSS-Fuzz files (*Dockerfile, build.sh and project.yaml*) and a fuzzing harness. Each of these integrations can be built and run locally using the OSS-Fuzz setup.

The mechanics behind OSS-Fuzzgen are divided into five sequential stages, and these five stages happen for each open source project input to OSS-Fuzzgen:

- **Stage 1: Build system generation.** This stage creates a build system comprising the OSS-Fuzz Dockerfile and build.sh to build the target codebase. The challenge of this stage is to automatically build a Java project purely based on the GitHub URL.
- **Stage 2: Target project static analysis.** This stage uses static program analysis to extract details, such as method signatures, of the target code which can be used for fuzzing harnesses generation. This stage relies on building the target code and performing static program analysis during the building.
- **Stage 3: Fuzzing harness generation.** This stage takes as input the data generated from Stage 2, and uses it to generate a candidate set of fuzzing harnesses. These harnesses are Java source code files that can be linked to the target's project build artefacts.
- **Stage 4: Fuzzing harness validation.** This stage combines the output from stage 1 and stage 3 into a set of candidate OSS-Fuzz project integrations and then builds and runs the fuzzing harness for each candidate project. The output of this stage is a set of logs showing the result of running the fuzzing harness for each candidate integration.

- **Stage 5: Fuzzing harness integration ranking.** This stage interprets the output from stage 4 and ranks each of the candidate OSS-Fuzz integrations. The output of this stage is a list of viable OSS-Fuzz integrations that are ranked according to which is the best integration.

IV. OSS-FUZZGEN DESIGN

This section describes the stages of OSS-Fuzzgen in detail, including implementation details and higher-level design decisions.

A. Stage 1: Build system generation

The first stage generates the OSS-Fuzz *Dockerfile* and *build.sh*, which are used to build the project in the OSS-Fuzz container image. The general problem to be solved is how to build a given arbitrary Java project and instructions for building fuzz harnesses against the project's build artefacts.

Java projects can be built in many different ways, such as directly compiled by *Javac* or using managed build systems like Maven or Gradle. To this end, OSS-Fuzzgen supports three build systems Maven, Gradle and Ant. OSS-Fuzzgen has heuristics for recognizing which build system is used by the target project by traversing the files of the target repository in the search for build files related to each build system. Specifically, OSS-Fuzzgen looks for *pom.xml* for Maven, *build.gradle* or *build.gradle.kts* for Gradle and *build.xml* for Ant. If multiple build properties exist, it indicates that the project can be built using multiple different build systems, OSS-Fuzzgen will use the first supported build system from the order: Maven, Gradle, Ant.

In addition to the build system, OSS-Fuzzgen needs to support different versions of the Java Development Kit (JDK). The default JDK version adopted by OSS-Fuzz is OpenJDK-15 at the time of writing. However, many projects require a different version of JDK to compile. To support this, OSS-Fuzzgen tries building the project using different versions of JDK until a successful build is found. The order of the JDK used are OpenJDK-15, OpenJDK-17, OpenJDK-11 and OpenJDK-8 and OSS-Fuzzgen will record and use the first successful build.

Finally, in addition to the build system and JDK version, an important step is identifying the class and jar files produced by the project, as these are necessary when linking fuzzing harnesses to the code. To support this, OSS-Fuzzgen traverses the folder of the project post-building to find the class files generated by the build and packs these class files into a single jar file. Additionally, OSS-Fuzzgen locates possible project jars, including dependencies, and moves them to a suitable classpath location so the generated fuzzing harnesses can use them.

B. Stage 2: Target project static analysis

The next task is to extract information about the target code for generating fuzzing harnesses. To do this, OSS-Fuzzgen relies on the Java frontend of Fuzz Introspector to retrieve a list of methods and classes of the target project. The list includes type information for each function, including both return type and argument types.

The Java frontend logic analyses the project's class and jar files, including third-party dependencies. However, OSS-Fuzzgen is not interested in generating harnesses for third-party dependencies, and, therefore, limits the analysis to the code within the source code directory of the target project. This is achieved by introspecting the source code location of the methods and classes within the jar files.

The static analysis component depends on the Soot framework, and a limitation of this is that the Soot framework fails to discover generic types and lambda expressions in the target code. For this reason, OSS-Fuzzgen can only generate general parameters for methods requiring generic type parameters or lambda expressions as input.

Following the static program analysis step, OSS-Fuzzgen creates a base OSS-Fuzz project integration directories and generates the correct set of base files from the template and the build configurations obtained in stage 1. OSS-Fuzzgen also includes an empty base fuzzing harness in the directory. At this point, OSS-Fuzzgen has created a Dockerfile, build.sh and a fuzzing harness, although the fuzzing harness is empty. The setup can now be tested in the OSS-Fuzz container images.

C. Stage 3: Fuzzing harness generation

This stage uses the output of the static analysis stage to create fuzzing harness source codes and combine them with the build artefacts from stage 1 to create working OSS-Fuzz integrations. To do this, OSS-Fuzzgen uses three steps to transform the raw data from Fuzz Introspector to a set of candidate OSS-Fuzz integrations, each with a fuzzing harness targeting the project.

The first step is extracting the specific methods in the target code to add metadata describing how to call these methods. Fuzz Introspector iterates through all the possible methods and classes in the project where each method may require different handling to execute. For example, some methods may be declared static which allows direct invocation, and some methods may require object creation or other code initialization. Furthermore, some methods may be class constructors or throw exceptions that need specific handling. This step extracts this information from the Fuzz Introspector result and groups the target methods accordingly.

The second step is filtering methods to reduce the candidate set of target methods. It is not uncommon for a medium-sized Java project to have more than a thousand methods, where many of them are not good targets to fuzz. OSS-Fuzzgen applies four filters to discard non-relevant methods:

- **Inaccessible methods filter.** This filter discards inaccessible classes and methods. This includes abstract classes, interfaces or protected / private elements which are not accessible by fuzzing harnesses and will likely fail in the fuzzing harnesses validation stage.
- **Helper methods filter.** This filter discards methods that do not have a lot of complexity. This includes general

methods from the Object class, methods with no parameters or helper methods that only get or set variables.
- **Out-of-scope methods filter.** This filter discards methods that do not belong to the target project. Some projects include third-party dependencies in their resulting jar files. OSS-Fuzzgen identifies the source code location for the target project and filters out all methods and classes which are not part of the source files for the target project.
- **Method call-depth filter.** This filter discards methods that may be hit by other possible entry points. Specifically, OSS-Fuzzgen extracts the call tree of each method and discards methods if other possible entry points will reach the given method. This filter consists of two stages. The first stage sorts all target methods by calling tree depth descendingly. Target methods with deeper call trees likely cover more logic which is a desired property when fuzzing. OSS-Fuzzgen then keeps the top 20% of the sorted method target list. The second stage adds any methods that are not called by any other methods, as these are considered public APIs which are determined to be good candidates for fuzzing.

Following the filtering step, OSS-Fuzzgen now has the list of method candidates to target and metadata on how to invoke each of these methods. Next, OSS-Fuzzgen proceeds to apply 10 heuristics for creating fuzzing harnesses against the target methods. These heuristics create a fuzzing harness that calls the target method in a manner where the arguments to the method are seeded with fuzzer-provided data. Some of these heuristics may produce code that won't run for a given target method.

The idea behind this step is to generate a lot of potential fuzz harness candidates and then use runtime evaluation later in the process to assess the quality of each harness. These heuristics are summarised in Table I. We came up with these heuristics by studying the existing OSS-Fuzz projects and abstracting existing fuzzing harnesses into higher-level code patterns.

In addition to creating the logic around the heuristics, OSS-Fuzzgen adds possible exception handling by traversing the call tree of each method, as well as including the import statements necessary for the code. OSS-Fuzzgen augments the code with comments that indicate the target methods and heuristics used.

Heuristics 1-4 are simple heuristics that consider different ways to execute static methods and instance methods directly. Static methods can be invoked directly while instance methods require object initialisation. For these four heuristics, OSS-Fuzzgen handles methods with up to 20 parameters where the parameters have to be primitive types, an array of primitive types and String (or CharSequence in general). Each argument is seeded with data from the fuzzing engine.

Heuristics 6-10 are more complicated than heuristics 1-4. Heuristic 6 considers some method execution that requires prerequisite settings and auto-discover possible settings methods and invokes them before the target method is executed. Heuristic 7 considers testing the consistency of method calling of some supposedly deterministic method. Heuristic 8-10 considers complicated parameters in addition to simple object creation, primitive types, an array of primitive types and string considered in heuristic 1-4. Those complicated parameter types include Class object, Enum object and parameters that require a fixed set of choices. Last but not least, heuristic 11 considers various kinds of parameters for executing public and concrete class constructors.

TABLE I. HEURISTICS FOR GENERATING FUZZING HARNESSES

Heuristic 1	Each possible target contains a fuzzing harness calling to one of the static methods in the target method list directly.
Heuristic 2	Each possible target contains a fuzzing harness calling to one of the instance methods in the target method list after the creation of the required object with the object constructor. It will search for a constructor from the subclass if the target object is an abstract class or interface.
Heuristic 3	Each possible target contains a fuzzing harness calling to one of the instance methods in the target method list after the creation of the required object using a static method like get instance or else.
Heuristic 4	Each possible target contains a fuzzing harness calling to one of the instance methods in the target method list after the creation of the required object with static or instance factory methods. It will also create an instance of the class containing the factory methods if necessary.
Heuristic 6	Similar to Heuristic 2-4, but before the target method is called, some setting methods will be called to simulate the case that some methods have some prerequisite method before the real execution logic.
Heuristic 7	Similar to Heuristic 2-4, but it will execute the target method twice and compare the result to fuzz for a deterministic result.
Heuristic 8	Similar to Heuristic 2-4, but it will handle enum type parameters with random choice of enum value.
Heuristic 9	Similar to Heuristic 2-4, but it will handle parameters that request a static number of choices.
Heuristic 10	Similar to Heuristic 2-4, but it will handle class type parameters of the target method.
Heuristic 11	Each possible target contains a fuzzing harness calling to one of the class constructors from classes in the project, excluding throwable classes or test classes.

The result of this stage is a set of candidate fuzzing harnesses where each of them is stored in an OSS-Fuzz integration directory together with the generated Dockerfile, build.sh and project.yaml. At this point, each of these directories represents a candidate OSS-Fuzz project.

A sample fuzzing harness is shown in Figure 1. The target method in this example is *feign.template.UriUtils::encode* and the heuristic applies is Heuristic 1. The heuristic simply calls into the static method using arguments seeded with data from the fuzz engine.

D. Stage 4: Fuzzing harness validation

Following the fuzzing harness generation, OSS-Fuzzgen has assembled a list of possible fuzzing harness integrations. OSS-Fuzzgen then validates each fuzzing harness by building and running it using the wrapping OSS-Fuzz project integration. The output from the runtime execution is logged and the return value and messages are used to judge if the execution

```
import com.code_intelligence.jazzer.api.FuzzedDataProvider;
import feign.template.UriUtils;

// jvm-autofuzz-heuristics-1
public class Fuzz {
  public static void fuzzerTestOneInput(FuzzedDataProvider data) {
  // Heuristic name: jvm-autofuzz-heuristics-1
  // Target method: [feign.template.UriUtils] public static java.lang.String
  //                  encode(java.lang.String,boolean)
  feign.template.UriUtils.encode(data.consumeString(100),data.consumeBoolean());
  }
}
```

Figure 1. Sample fuzzing harness generated by OSS-Fuzzgen on feign.template.UriUtils::encode method of project feign using heuristic 1

is successful or not. The runtime execution time can be set by the user of OSS-Fuzzgen and is by default set to 20 seconds.

OSS-Fuzzgen determines the status of the run with some additional fuzzing statistics including coverage information. These logs are stored in a separate directory together with a summary.json recording key statistical data for later analysis purposes. Both the building and running of the harness may break, either due to limitations in the artefacts produced, SOOT, Fuzz Introspector or the generated code.

E. Stage 5: Fuzzing harness integration ranking

The OSS-Fuzzgen results are ready to use after the fuzzing harnesses validation phase, however, OSS-Fuzzgen may have generated several hundred successful runs for any given project. To aid the analysis and choosing of the best result to be integrated into OSS-Fuzz, OSS-Fuzzgen also provides some post-processing and summarization of data. OSS-Fuzzgen ranks the possible targets according to the maximum code coverage achieved and the maximum difference in coverage between the start and finish of each fuzzing run.

OSS-Fuzzgen also comes with several utilities for extracting an overview when analysing many open source projects at the same time, to ease the efforts needed to identify the best performing harnesses. The resulting OSS-Fuzz integration directories for each successfully generated target can be integrated directly into OSS-Fuzz.

V. EMPIRICAL STUDY OF OSS-FUZZGEN PROCESS AND GENERATED FUZZING HARNESSES

In this section, we present the empirical evaluation of our work. The evaluation process consists of two experiments: a large-scale study running OSS-Fuzzgen autonomously and an extension of this study where we integrate a subset of the successful projects into OSS-Fuzz with minor manual additions.

A. Large scale evaluation

To empirically verify OSS-Fuzzgen, we ran it against 257 open source Java projects not covered by OSS-Fuzz yet. We made an effort to pick popular Java libraries or frameworks, where popularity was based on the number of GitHub Star and GitHub Watch rankings. There were no UI applications in the target projects and in general, we picked libraries that are meant for use by applications rather than stand-alone applications as such. To set up the experiment, we created a text file containing the URLs to each of the 257 projects and provided it as input to OSS-Fuzzgen. For the validation phase of OSS-Fuzzgen, we set the fuzzing harnesses to run for 20 seconds. We divide the results into the following categories:

1) **S1: Successful build and generate fuzzing harness.** A build script and some fuzzing harnesses were generated. Fuzzing harnesses may not be runnable.
2) **S2: Successful build and generate fuzzing harness that runs.** A build script and fuzzing harnesses were generated. Some fuzzing harnesses are built and run successfully.
3) **S3: Successful build and generate fuzzing harness that runs and increases coverage.** A build script and fuzz harnesses were generated. Some fuzzing harnesses build and run successfully and explore more than one code path within 20 seconds of execution.

TABLE II. RESULTS FROM RUNNING OSS-FUZZGEN ON OPEN SOURCE SOFTWARE

#	Total java project targets	257	100%
S1	Successful build and generate fuzzing harness	123	47%
S2	Successful build and generate fuzzing harness that runs	116	45%
S3	Successful build and generate fuzzing harness that runs and increases coverage	94	37%

Table II shows the results of our evaluation. Amongst the 257 total targets, OSS-Fuzzgen succeeded in generating project integrations that match category S2 for 116 (45% of the total) projects. However, 22 of these generated projects failed to increase coverage during the initial 20 seconds of fuzzing harnesses validation, meaning a total of 94 projects (37% of the total) got results matching group S3.

B. Submitting projects to OSS-Fuzz

The goal of OSS-Fuzzgen is to generate OSS-Fuzz integrations that are useful in testing and fuzzing the code of the target projects. To empirically validate this goal, we submitted 31 of the 94 resulting OSS-Fuzz integrations where we picked those projects with the most promising signs of code exploration. We identified this by looking at the code coverage delta of the harnesses achieved from the 20-second initial fuzzing run. The goal was to monitor if the fuzzing harnesses found any issues in the target projects and ensure the projects ran continuously.

Before submitting the generated projects to OSS-Fuzz, we applied some manual efforts on several resulting projects. First, when OSS-Fuzzgen generated multiple targets for a given project, we hand-picked the best targets, in terms of code coverage and target method call depth, and merged them into a single directory. Second, sometimes the auto-generated code may reveal additional entry points in the target code that are good for fuzzing. For example, additional functions that are fuzzable within the same class as an auto-generated fuzzing harness, and we added these. Third, some of the promising generated harnesses would run into issues early in the execution due to missing initialization code and in these cases, we added logic to the fuzzing harnesses that would properly initialise the relevant logic. Finally, we went over the auto-generated code to improve readability by e.g., setting the names of variables appropriately and cleaning up code formatting.

Amongst the 31 projects we submitted to OSS-Fuzz, we received more than 50 bug reports. Commonly, projects with issues have 2 to 3 issues reported whereas a few projects have a significantly higher amount. For example, Joni has 7 bugs reported, however, after root-cause analysis we found that they are caused by triggering 2 core bugs via different entry points, meaning the two bugs are triggered in a handful of ways. The types of bugs found include out-of-memory errors, integer overflow errors, regular expression Denial-of-Services, index out-of-bounds errors for array or string accesses, and string encoding errors.

TABLE III. UPSTREAM BUG FIXING STATUS

Projects	# bug fixes	Status
https://github.com/fusesource/jansi	2	Accepted
https://github.com/jruby/joni	2	Accepted
https://github.com/openfeign/feign	2	Accepted
https://github.com/virtuald/curvesapi	1	Accepted
https://github.com/xdrop/fuzzywuzzy	1	Accepted
https://github.com/graphql-java/graphql-java	1	Accepted
https://github.com/fasseg/exp4j	1	Submitted
https://github.com/locationtech/jts	1	Submitted

To verify that the issues found by the fuzzing harnesses are valid, we performed a root-cause analysis of 11 of these from 8 different projects. Most of the bugs are invalid input checking or memory overflow issues. We then generate bug fixes and make pull requests with fixes on the relevant repositories. 9 bugs from 6 projects have been accepted and merged by the project maintainers with positive comments. Table III summarises the bug reports.

VI. LIMITATION AND FUTURE WORK

OSS-Fuzzgen has several limitations in the implementation domain. First, extending the system to support more build systems and more versions of JDK would enable more targets to be processed. For almost half the projects tested OSS-Fuzzgen created an OSS-Fuzz integration that builds the project with a fuzzing harness that runs. Extending to further JDK and build systems will likely increase this proportion.

Additionally, we can extend the system to support lambda expressions and generic types for fuzzing harness generation. To do this, we can migrate the existing Fuzz Introspector frontend with SootUp[17].

A limitation in OSS-Fuzzgen is the scope of generating fuzzing harnesses. Currently, it's limited to 10 different heuristics. We can extend this to support additional heuristics to increase the possible set of fuzzing harnesses to generate.

An interesting avenue for improving fuzzing harness generation is extending the system with more general approaches. For example, recent work has explored using Large Language Models to generate fuzzing harness code which shows promising results[18].

To integrate the projects in OSS-Fuzz, we picked the best projects constructed by OSS-Fuzzgen based on how much a given integration achieved in code coverage exploration. A limitation is that we made manual assessments in this case, and further work would explore how we can improve the ability to rank the auto-generated projects. This includes features such as automatically identifying the threat model of a project and matching this with auto-generated fuzzing harnesses; automatically assessing how security-critical an open source project is to enable selection of those where vulnerabilities are most important and also include more data from Fuzz Introspector as to how promising a given harness is.

VII. CONCLUSION

In this paper, we introduce OSS-Fuzzgen, a first effort in automatic OSS-Fuzz project integration. OSS-Fuzzgen enables automatic fuzzing of open source Java projects by generating fuzzing harnesses, constructing appropriate build scripts and validating the generated harnesses to identify those that perform the best.

OSS-Fuzzgen significantly lowers the barrier of entry for continuous fuzzing and to demonstrate these capabilities, we ran OSS-Fuzzgen against a dataset of 257 open source Java projects. As a result, OSS-Fuzzgen produced 116 valid OSS-Fuzz project integrations with some fuzzing harnesses that build and run. Furthermore, to prove the use of the generated fuzzing harnesses we added 31 of these projects to OSS-Fuzz which resulted in more than 50 issues being found. Finally, we submitted bug fixes for 11 reported issues, 9 of which have been accepted and merged by the open source projects.

In conclusion, OSS-Fuzzgen provides a good entry point for open source project fuzzing. This setting could encourage open source project maintainers to start fuzzing their projects and adopt OSS-Fuzz for continuous security issues and bug discovery.

ACKNOWLEDGMENT

We would like to thank the OSS-Fuzz team for responding to our issues on GitHub and reviewing our contributions. We would also like to thank the maintainers who reviewed our bug fixes.

REFERENCES

[1] "OSS-Fuzzgen." https://github.com/AdaLogics/OSS-Fuzzgen, 2023. Retrieved: October, 2023.

[2] "OSS-Fuzz." http://github.com/google/oss-fuzz, 2023. Retrieved: October, 2023.

[3] Z. Y. Ding and C. L. Goues, "An empirical study of OSS-Fuzz bugs," *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 131–142, 2021.

[4] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, pp. 2312–2331, nov 2021.

[5] "Fuzzing Introspection of OSS-Fuzz projects." https://introspector.oss-fuzz.com/, 2023. Retrieved: October, 2023.

[6] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "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*, 2019.

[7] Y. Fu, J. Lee, and T. Kim, "autofz: Automated fuzzer composition at runtime," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023* (J. A. Calandrino and C. Troncoso, eds.), USENIX Association, 2023.

[8] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (S. Capkun and F. Roesner, eds.), pp. 2271–2287, USENIX Association, 2020.

[9] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pp. 2676–2692, IEEE, 2023.

[10] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), p. 213–223, Association for Computing Machinery, 2005.

[11] C. Rahalkar, "Automated fuzzing harness generation for library APIs and binary protocol parsers," 06 2023.

[12] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 371–382, 2009.

[13] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 288–298, 2012.

[14] "Fuzz Introspector." http://github.com/ossf/fuzz-introspector, 2023. Retrieved: October, 2023.

[15] W. Gao, V. Pham, D. Liu, O. Chang, T. Murray, and B. I. P. Rubinstein, "Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report)," in *Proceedings of the 2nd International Fuzzing Workshop, FUZZING 2023, Seattle, WA, USA, 17 July 2023* (M. Böhme, Y. Noller, B. Ray, and L. Szekeres, eds.), pp. 47–55, ACM, 2023.

[16] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, "The soot framework for java program analysis: a retrospective," October 2011. Event Title: Cetus Users and Compiler Infastructure Workshop (CETUS 2011).

[17] "SootUp, howpublished = https://soot-oss.github.io/sootup/, year = 2023, note = Retrieved: October, 2023."

[18] "Fuzz target generation using LLMs." https://google.github.io/oss-fuzz/research/llms/target_generation/, 2023. Retrieved: October, 2023.