VeriFuzz: Program Aware Fuzzing
(Competition Contribution)

Animesh Basak Chowdhury, Raveendra Kumar Medicherla(✉), and Venkatesh R

Tata Research Development and Design Centre, Pune, India
raveendra.kumar@tcs.com

Abstract. VeriFuzz is a program aware fuzz testing tool, which combines the power of feedback-driven evolutionary fuzz testing with static analysis. VeriFuzz deploys lightweight static analysis to extract meaningful information about program behavior that can aid fuzzing based test-input generation to achieve coverage goals quickly. We use constraint-solver to generate an initial population of test-inputs. VeriFuzz could generate the maximum number of counterexamples for `reachsafety` category benchmarks in SV-COMP 2019 and in Test-Comp 2019 [16]. (All the terms in typewriter font are competition specific. See [15].)

1 Introduction

VeriFuzz is a coverage driven automated test-input generation tool based on *grey-box* fuzzing [5]. The idea of grey-box fuzzing is to use lightweight instrumentation to observe behaviors exhibited during a test run. This information is used while fuzzing for new test-inputs that might exhibit new behaviors. For VeriFuzz, the behavior of interest is code coverage. VeriFuzz relies on evolutionary algorithms to generate newer test-inputs from an initial population of test-inputs. Central to an evolutionary algorithm is the *selection* of *best-fit* candidates from a *population* and generate offspring by applying *crossover* and *mutating* operations on them. The newer offspring are checked for their *fitness* against a goal. The population evolves by adding the fit offspring to the existing population. In an automated testing, a candidate test-input plays the role of an individual in a population. The new test-inputs are generated from a selected test-input by repeatedly applying mutation operations, for example, by flipping byte at a random position. The fitness of a generated test-input is determined by the code coverage during its run [11,18].

State-of-the-art grey-box fuzzers such as afl-fuzz [19], though simple to use, have several key shortcomings. (a) The fuzzer is aware of neither the program structure nor the input structure. This leads to the generation of a large set of redundant test-inputs with respect to code coverage. (b) For programs that

R. K. Medicherla—Jury Member.

D. Beyer et al. (Eds.): TACAS 2019, Part III, LNCS 11429, pp. 244–249, 2019.
https://doi.org/10.1007/978-3-030-17502-3_22

does *complex* validations on their input, the fuzzer finds it hard to generate a test-input that satisfies such validation conditions [13]. Finding a suitable initial population of test-inputs for such programs requires the analysis of validation conditions. (c) For programs with unbounded loops, the fuzzer may get *stuck* forever without generating any new test-inputs. There are several approaches proposed in the literature to address some of these shortcomings [1,7,12,13]. However, all these approaches address each concern separately.

2 Our Approach

In order to alleviate the problems described in Sect. 1, our approach analyses and transforms the subject program. The analysis information is then passed to the enhanced mutation engine of afl-fuzz for fuzzing the transformed program. The following are the key steps of our approach.

Efficient Instrumentation: To measure the coverage due to a run on a test-input, the subject program is *instrumented*. However, instrumentation adds a significant overhead to the program execution, impacting the fuzzer's execution speed. We have optimized the instrumentation overhead by placing the probes either true or false branches of each conditional statement in the program. Our scheme is efficient to implement and preserves the coverage measure though it is sub-optimal than instrumentation schemes proposed in literature [7].

Loop Bounding: Certain class of programs, for example, reactive programs, during their execution, either does not terminate or crash upon reaching the *error location*[1]. In order to handle such non-terminating programs, our approach transforms the program loops by replacing the condition in their loop heads with a known bound. This bound is increased dynamically during the fuzzer run till it finds an input that can take program execution to an error location or the budgeted time elapses.

Novel Initial Test Population Generation: Grey-box fuzzers find it hard to generate test-inputs that can take program execution to cover the program blocks that are guarded by complex checks [10,13]. If the initial test-input population can take program execution through some of the complex checks, fuzzing such inputs is *likely* to generate test-inputs that can pass through other complex checks [17]. In order to create such initial test-input population, our approach first flattens the program by unrolling the loops up to a certain bound. Then, a program path is chosen that contains such complex checks and path constraints are generated along that path. The constraints are solved to create an initial test-input population.

Program Analysis to Assist Mutation: For programs that read input only within restricted range values, it is possible to fine-tune the fuzzer's mutation operators to choose values within this restricted ranges. In order to determine

[1] Program statement where error function `__VERIFIER_error()` is called.

the ranges of input values that can reach the error locations, our approach statically determines the input value ranges of the program using k-path interval analysis [9]. For a given program, this analysis determines the conservative over approximate ranges of input values that may reach any given program point. We have enhanced the mutation engine of the fuzzer such that it accepts the input value ranges at the error location in the program and generates inputs that have values within the given ranges.

Algorithmic Selection of Strategies: All the aforementioned techniques are generic enough to use across the programs. However, in order to optimize the given resource budgets in the competition, we have selectively applied a subset of techniques to a specific class of programs. For example, loop bounding technique is applied to programs where syntactic unbounded loop structures are detected. In order to identify and map the best performing set of techniques to all benchmark programs, we have grouped them into a finite set of classes and formulated it as *multi-label classification* problem. The classifier model is developed using a non-parametric supervised learning based approach [6]. The model has been trained using nine syntactic structures of a C program and a subset of techniques as classification labels. The benchmark programs from SV-COMP 2018 [14] were used as training and validation set. We have used the decision tree classifier for this multi-label classification [2].

3 Tool Architecture and Flow

Figure 1 shows the architecture of the VeriFuzz tool. It consists of a fuzzing engine and an analysis engine. The core fuzzing engine is built on top of the state-of-art grey-box fuzzer afl-fuzz v2.52b [19]. The program analysis, instrumentation and transformation components of the analysis engine are implemented using the PRISM, a TCS in-house program analysis framework [8]. The initial input generation component uses CBMC v5.10 [3] as path-constraint solver. The program classification component uses an offline trained model and scikit-learn v.0.19.2 to access the model. The implementation is in C, Java, and Python languages.

The input to the tool is a program P and a safety property ϕ. In the first step, the syntactic features are extracted and the class of the program is determined using a program classification module. This class information is used in subsequent steps. In the second step, the program P_i is generated using an instrumentation and transformation module. This step also emits the transformed programs for `witness generation` (P_w) and initial test-input generation modules. In the next step, the program is analysed to determine input ranges. These input ranges are used to formulate the fuzzing engine parameters F_i. Subsequently, initial test-input population T_i is generated using the initial input generation module. The fuzz engine is then invoked with P_i, F_i, and T_i as inputs.

As a first step of the fuzzing engine, the program P_i and a harness program that implemented `__VERIFIER_*` functions are compiled together using `gcc` to generate the executable program P_e. The core fuzzer begins with T_i as its initial

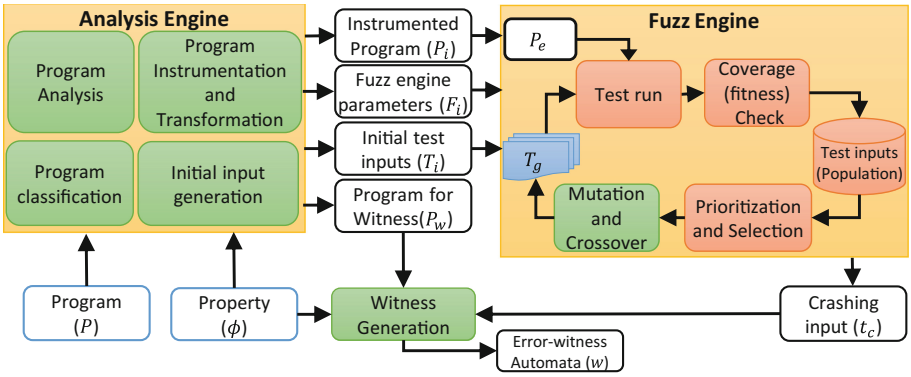

Fig. 1. VeriFuzz architecture.

population, executes P_e, and measures the coverage. A test-input from the population is selected and mutated several times to generate newer test-inputs T_g. The program P_e is repeatedly executed with each test-input $t_g \in T_g$ and coverage is measured. The fitness check step compares the code coverage due run on each t_g with the historical code coverage and determines whether t_g should be added to the population or not. This process is repeated until the core fuzzer finds a crashing test-input t_c that causes the program run to reach the error location or the time budget is elapsed. The t_c and P_w are passed to a witness generation program to generate `error-witness`.

4 Strengths and Weaknesses

The core strength of VeriFuzz is its ability to find a test-input that can cause the program execution to reach the error locations quickly. The tool participated both in SV-COMP and Test-Comp [16]. In SV-COMP, VeriFuzz could identify 1264 out of 1458 `reachsafety FALSE` benchmarks with 67 s as mean time per `verification task`. Whereas in Test-Comp, it could identify `bugs` in 592 out of 636 benchmarks. VeriFuzz could generate test-inputs that can, on an average, cover 70% branches in 1720 benchmarks.

VeriFuzz explores the concrete program paths randomly and redundantly due to its evolutionary approach. Therefore it may not always discover a test-input that cause the execution to reach an error location.

5 Tool Configuration and Setup

The VeriFuzz tool for testing SV-COMP benchmarks is available at the URL https://gitlab.com/sosy-lab/sv-comp/archives-2019/blob/master/2019/verifuzz.zip. Its Test-Comp 2019 variant is available at the URL https://gitlab.com/sosy-lab/test-comp/archives-2019/blob/master/2019/verifuzz.zip. The `benchexec`

tool-info module is verifuzz.py and the benchmark description file is verifuzz.xml. To install and run the tool, follow the instructions provided in README.txt with the tool. A sample run command is as follows:

```
./scripts/verifuzz.py --propertyFile unreach-call.prp example.c
```

6 Software Project and Contributors

VeriFuzz is developed by the authors at TCS Research. We would like to thank B. Chimdyalwar and S. Kumar from VeriAbs [4] team for the help in the understanding of k-path interval analysis.

References

1. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 2329–2344. ACM (2017)
2. Chen, T., Guestrin, C.: XGBoost: a scalable tree boosting system. In: Proceedings of the 22nd SIGKDD International Conference on Knowledge Discovery and Data mining (KDD), pp. 785–794. ACM (2016)
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
4. Darke, P., et al.: VeriAbs: verification by abstraction and test generation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 457–462. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_32
5. DeMott, J., Enbody, R., Punch, W.F.: Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. BlackHat and Defcon (2007)
6. Demyanova, Y., Pani, T., et al.: Empirical software metrics for benchmarking of verification tools. Form. Meth. Syst. Des. **50**(2–3), 289–316 (2017)
7. Hsu, C.C., Wu, C.Y., Hsiao, H.C., Huang, S.K.: INSTRIM: lightweight instrumentation for coverage-guided fuzzing. In: Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research (2018)
8. Khare, S., Saraswat, S., Kumar, S.: Static program analysis of large embedded code base: an experience. In: Proceedings of the India Software Engineering Conference (ISEC) (2011)
9. Kumar, S., Chimdyalwar, B., Shrotri, U.: Precise range analysis on large industry code. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 675–678. ACM (2013)
10. Lemieux, C., Sen, K.: FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 475–485. ACM (2018)
11. McMinn, P.: Search-based software testing: past, present and future. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 153–163. IEEE (2011)
12. Rawat, S., Jain, V., et al.: VUzzer: application-aware evolutionary fuzzing. In: USENIX security (2017)

13. Stephens, N., Grosen, J., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2016)
14. SV-COMP 2018 Benchmarks: (Commit - f2996ff). https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp18
15. SV-COMP, Test-Comp: Definitions and Rules (2019). https://sv-comp.sosy-lab.org/2019/rules.php, https://test-comp.sosy-lab.org/2019/rules.php
16. TOOLympics 2019: Competetion on software testing (Test-Comp). TACAS 2019 (2019). https://test-comp.sosy-lab.org/2019/
17. Wang, J., Chen, B., Wei, L., Liu, Y.: SkyFire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 579–594. IEEE (2017)
18. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Inf. Softw. Technol. **43**(14), 841–854 (2001)
19. Zalewski, M.: American fuzzy lop. http://lcamtuf.coredump.cx/afl/