Towards Fuzzing Target Lines

Nuno Neves
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
nuno@di.fc.ul.pt

***Abstract*—Fuzzers are often considered one of the most effective tools to uncover bugs in software, including security vulnerabilities. Current tools are designed to test the full programs by employing strategies that select and generate testcases that increase code coverage. In this paper, we propose to evolve these ideas to make fuzzers focus on specific portions (i.e., lines) of the code, which we call *targets*, avoiding wasting resources on the other regions. The aim is to support efficient testing of evolving software, and other practical scenarios such as confirming the results from static analysis.**

Keywords—Testing, Target Fuzzing, Vulnerability Discovery

I. Introduction

Security vulnerabilities in software allow malicious actors to subvert normal operation, often with dire consequences, including loss of property and reputational damage. According to Common Vulnerabilities and Exposures (CVE), the number of new entries surpassed 55.000 over the period 2019-21 [1]. When organized by type, several of the most prominent classes are associated with bugs usually found in software programmed with the C/C++ languages, such as overflows, memory corruptions, and arbitrary code execution. Moreover, C/C++ is regularly the choice for critical application development, and therefore, it is of utmost importance to expand the vulnerability discovery capabilities in these languages.

Fuzzing is highly effective at checking complex codebases, usually being able to discover flaws in programs previously analyzed with other techniques. A coverage-guided fuzzer operates in a conceptually straightforward manner, following an approach akin to a genetic algorithm. The fuzzer executes iteratively in a loop. It begins with a few user-supplied inputs (or testcases), which are provided to the program under test in independent executions. Feedback data is captured at runtime allowing the fuzzer to recognize coverage-gaining inputs (e.g., additional statements/basic blocks are reached), which are retained for the next phase. These inputs are also evolved (various mutation and crossover strategies are applied) to produce many offspring, and the loop starts over again with a new generation of testcases. While running, the fuzzer searches for program misbehaviour (e.g., out-of-bounds memory access) as an indication that a bug was activated. The corresponding input is then given to developers facilitate debugging activities.

Fuzzing is, however, extremely slow. For example, a recent fuzzer took 60 days of testing to find a few tens of bugs in databases [2]. A benchmark recently published showed that even after spending over 200 000 CPU-hours, only 46% of the injected vulnerabilities (54 in 118) were detected by a group of 7 state-of-the-art fuzzers [3]. These fuzzers operate by checking the whole program, as the aim is to extend testing to as many statements as possible. However, this is not appropriate if one needs to test evolving software, which comes into play in many practical scenarios, such as when a new release is created, when a patch needs to be assessed, to verify reports from static analysis tools, and to reproduce previously observed failures. Here, fuzzing should focus on certain sections of the code, to avert exhausting the available time for testing with superfluous analyses.

Our goal is to design and implement a fuzzer that when given a list of lines of code, which we call *targets*, focusses its energy in reaching those lines with appropriate input testcases, and then tests them (and the lines around) exhaustively.

II. Challenges

A. *Diversity of scenarios*

Depending on the testing scenario, the targets can appear in the code in very different manners. For example, they can form a cluster of statements (e.g., a function was changed and needs to be tested) or they can be spread over various files of the program; they can be composed by a small group of instructions (e.g., a patch or a particular line identified by static analysis) or can correspond to tens of thousands of lines of code (e.g., checking modifications before a new release). Independently, the goal is to efficiently find inputs that can reach the targets, so that they can be evolved by the fuzzer, to check those lines with many combinations of values, eventually triggering a bug.

B. *Determining if inputs are progressing towards targets*

Existing coverage-guided fuzzers employ highly optimized data structures to identify inputs that lead to novel program coverage. The ability to perform this selection is of fundamental value because it allows the fuzzer to distinguish inputs that are worthy to continue mutating. These data structures are, however, unable to determine if inputs are allowing the execution to get closer to the targets. In addition, since one would like to test all targets, it would be necessary to find out which testcases are progressing towards each particular target (as most probably, if targets are spread through the code, one will need to distinct inputs to reach the various parts).

C. *Keep the performance of fuzzing*

To address the previous challenge, one will require program instrumentation that collects more detailed information about which parts of a program are executed when processing an input. This instrumentation will introduce delays, decreasing the number of tests that can be performed per unit-of-time, and therefore, the performance of fuzzing. We expect, however, to offset these overheads by concentrating the instrumentation on the paths to the targets. This way we will eliminate these costs from all parts of the code that are irrelevant to get to the targets.

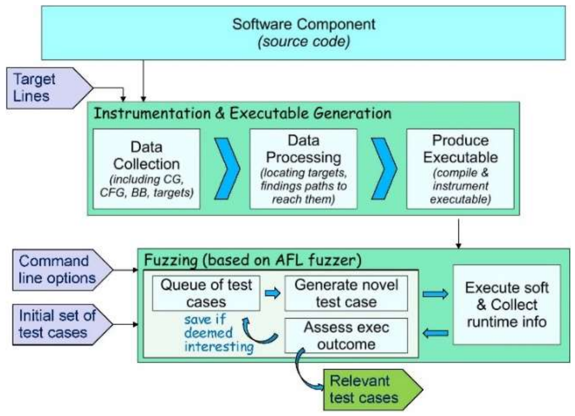
Software Component
(source code)
Target Lines
Instrumentation & Executable Generation
Data Collection (including CG, CFG, BB, targets)
Data Processing (locating targets, findings paths to reach them)
Produce Executable (compile & instrument executable)
Command line options
Initial set of test cases
Fuzzing (based on AFL fuzzer)
Queue of test cases
save if deemed interesting
Generate novel test case
Assess exec outcome
Execute soft & Collect runtime info
Relevant test cases

Fig. 1. Architecture of the fuzzer.

III. Current design and implementation

Our current design and implementation aim at testing large C/C++ applications (up to 1 million lines of code) for common bugs, such as memory management flaws. The prototype leverages the CLANG/LLVM compiler framework to collect information and instrument the programs. Various auxiliary tools are also being constructed, for example to recognize the lines that change among software releases and to create runtime feedback data. As a starting point for fuzzing support, we are using the AFL fuzzer, the most used open-source tool.

Figure 1 represents the main components of our design. The fuzzing process is divided in two parts. A first one where the source code is analyzed and instrumented, and finally an executable is created. The second part where the actual fuzzing occurs. In more detail:

a) *Data collection*: build a program analysis mechanism that produces an expressive graph representation augmented with the location of the targets. In C/C++, it is unfeasible to produce an exact graph as there are edges that may only be determined at runtime. Our aim is to be precise within these constraints, dealing appropriately with aspects like dynamic function pointers, indirect jumps, and inline statements.

b) *Data processing*: implements an algorithm to reason over the graph, locating diverse paths and computing distances from the program entry points to the targets, which is scalable to many nodes and edges. We are exploring the concept of *target clusters* to decrease complexity, where targets near to each other are aggregated. Most computation is done during compilation (offline) to minimize runtime overheads.

c) *Produce executable*: construct an instrumented software executable so that when it runs data is provided to the fuzzer about the parts of the code that are being reached. The instrumentation is only added to the paths towards the targets to make it highly efficient while still expressive enough.

d) *Fuzzing*: the fuzzer runs the software under different testcases, while exploring feedback data to discover which inputs are moving towards the targets.

The current prototype already implements the Instrumentation & Executable Generation, and currently we are developing the fuzzer. Once completed, we aim to test production-level software to assess its ability to discover relevant vulnerabilities. We will also compare our fuzzer with alternative solutions under various criteria, like number of targets found, progress in the direction of targets, and number of discovered vulnerabilities.

IV. Related Work

Among previous research that aims at security testing specific parts of the code, symbolic execution [4] is the one that has been applied more prominently. Although significant improvements have been observed over recent years, symbolic execution still suffers from several drawbacks, such as the number of paths to be explored can explode.

We are only aware of two fuzzers that attempt to reach a target, although with important limitations. AFLGO tries to steer the AFL fuzzer towards predefined program locations [5]. This is achieved by employing a simulated annealing heuristic to give extra energy to testcases that cause the execution to go near the selected locations, while reducing the energy of inputs that stay at a larger distance. Here, energy is related to the amount of time that is given for mutating a testcase into other inputs. One immediate downside of this approach is that it always favors a target for which it finds an input with the shortest distance, thus not dealing appropriately with multiple targets. Hawkeye tries to address this issue by keeping the testcases in a three-tiered queue [6]. Testcases in the top queues are given priority, namely those with smaller distances or that hit the targets. Although this solution is apparently better than AFLGO, it still suffers from many shortcomings. Examples are: (i) inputs that go through highly improbable path conditions are not identified, potentially preventing progress or causing a significant loss of effort; (ii) targets that may be hit through multiple paths are preferred at the cost of neglecting others; (iii) clusters of targets are ignored, thus constraining the scalability.

Acknowledgment

Work partially supported by FCT through project SEAL (PTDC/CCI-INF/29058/2017, LISBOA-01-0145-FEDER-029058, POCI-01-0145-FEDER-029058) and LASIGE research unit (UIDB/00408/2020 & UIDP/00408/2020), and by the European ITEA3 through project XIVT (I3C4-17039/FEDER-039238).

References

[1] CVEDetails: Vulnerabilities By Year, 2021 [Online: last accessed March 2022] https://www.cvedetails.com/browse-by-date.php

[2] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, D. Wu, "SQUIRREL: Testing database management systems with language validity and coverage feedback", In Proc. of the ACM Conf. on Computer and Communications Security, pp. 955–970, 2020

[3] A. Hazimeh, A. Herrera, M. Payer, "Magma: A ground-truth fuzzing benchmark", in Proc. of the ACM on Measurement and Analysis of Computing System, Vol. 4, No. 3, 2020

[4] C. Cadar, K. Sen, "Symbolic execution for software testing: Three decades later", Communications ACM, vol. 56, No. 2, pp. 82–90, 2013

[5] M. Böhme, V.-T. Pham, M.-D. Nguyen, A. Roychoudhury, "Directed greybox fuzzing", In Proc. of the ACM Conf. on Computer and Communications Security, pp. 2329–2344, 2017

[6] H Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer", In Proc. of the ACM Conf. on Computer and Communications Security, pp. 2095–2108, 2018