Efficient Fuzz Testing Leveraging Input, Code, and Execution

Nikolas Havrikov
Saarland University
Saarbrücken, Germany
havrikov@cs.uni-saarland.de

***Abstract*—Any kind of smart testing technique must be very efficient to be competitive with random fuzz testing. State-of-the-art test generators are largely inferior to random testing in real world applications. This work proposes to gather and evaluate lightweight analyses that can enable the creation of an efficient and sufficiently effective analysis-assisted fuzz tester. The analyses shall leverage information sources apart from the program under test itself, such as e.g. descriptions of the targeted input format in the form of extended context-free grammars, or hardware counters. As the main contributions, an efficient framework for building fuzzers around given analyses will be created, and with its help analyses will be identified and categorized according to their performance.**

I. Introduction

The ultimate goal of testing is to increase confidence in the application under test. As it is impossible to completely prove the absence of errors solely by means of testing, perhaps an even more desirable goal is to reveal defects in the application such that they can be mended, thus improving its quality. A standard technique for achieving this goal is automated test input generation also known as *fuzzing*.

Böhme et al. have shown that analysis-assisted test input generators are in practice often *less efficient* than random testing [1]. They state that if an analysis-assisted approach takes c time units for analyzing each input it produces and its effect on the program under test, there is a bound c_0 for the analysis cost, above which the approach is less efficient than random testing. These findings also find support in an investigation performed independently by Choudhary et al. who demonstrate the superiority of random testing over currently available fuzzers on the Android platform [2].

Unfortunately, it is infeasible to rely solely on random testing. Most applications require a specific input format and discard invalid inputs directly. Typically, the input validation itself is of little interest for testing; it does, however, present an obstacle for random input generators. It is extremely unlikely for an input generated randomly to fit a specific format and proceed deep into the business logic of the program under test. Some approaches (e.g. [3]–[6]) employ symbolic execution to extract information about these checks to produce valid inputs. Unfortunately, these approaches do not scale well with the size of the program under test. However, given a grammar describing the required input format, an input generator can walk along the productions of the grammar to compose a syntactically valid input much more efficiently.

```
URL ::= PROTOCOL '://' AUTHORITY PATH
        ['?' QUERY] ['#' REF]
AUTHORITY ::= [USERINFO '@'] HOST [':' PORT]
PROTOCOL ::= 'http' | 'ftp'
USERINFO ::= /[a-z]+/ ':' /[a-z]+/
HOST ::= /[a-z.]+/
PORT ::= '80'
PATH ::= /\/[a-z0-9.\/]*/
QUERY ::= 'foo=bar&lorem=ipsum'
REF ::= /[a-z]+/
```

Figure 1. Grammar extracted automatically from `java.net.URL` processing some inputs (taken from [7]). Optional elements are enclosed in **[...]** while regular expressions are signified by **/.../**.

Höschele et al. have shown a method for automatically extracting context-free grammars describing the input format from runs of an application using dynamic input tainting [7]. Figure 1 shows the grammar extracted from the `java.net.URL` class processing a set of sample inputs. Equipped with this grammar, a fuzzer can produce inputs capable of penetrating deeper into the URL-processing logic of any web browser. While the analysis used in the extraction of a grammar does not scale well, once extracted from one program, the same grammar can be used to generate test inputs for any other program that processes the same input format. Furthermore, it is possible to augment a grammar with additional constraints gathered by static analysis. For example, dependencies between specific parts of an input can be helpful in the creation of meaningful inputs that reach deep into the execution of a program.

Grammars provide feedback-driven fuzzers with information sources they can leverage in addition to telemetry from the program under test such as code coverage, which they use to reach certain *testing goals*. Such fuzzers usually use *metrics* on the telemetry to guide the generation of inputs toward the testing goals. For example, branch distance can be used as a guide to achieve a certain branch coverage goal.

Similarly, metrics can be defined on properties of the grammar. For example, some edit distance metric [8] can be used to guide the generation of inputs towards covering all alternatives of production rules in a grammar, thereby possibly exercising more functionality of the program under test. The presumably low cost of analyses required for the calculation of grammar-based metrics leads to research questions which I plan to address in my work:

RQ1 How does novel grammar-based fuzzing compare to traditional coverage-based approaches in terms of efficiency and effectiveness?

RQ2 How does novel grammar-based fuzzing compare to classic random input generation?

RQ3 Which metrics on grammars are best suited for guiding a fuzzer towards finding defects?

II. Background

Approaches for automatic test input generation can be roughly categorized based on whether they treat the program under test as a *black-box* or a *white-box* (i.e. gathering telemetry), as well as based on how much domain-specific knowledge about the input format they possess (e.g. a grammar description, some constant values, or none at all). Figure 2 provides a classification of some input generators according to these criteria.

A *black-box*, *random* technique was first developed by Miller et al. in 1990 – a fuzzer which produces inputs randomly without any knowledge of the input format and the program under test [9]. On the other end of the spectrum is *white-box* testing, which usually leverages format specific knowledge and application analysis to guide the production of inputs towards desired criteria like e.g. code coverage [10], data-flow coverage [11], or vulnerability pattern similarity [12].

The white-box input generator KLEE [5] uses symbolic execution to gather constraints along the execution path of a program and produces inputs that trigger this path by means of a SAT solver. The white-box approach SAGE by Godefroid et al. [6] uses dynamic symbolic execution to produce its inputs. Both approaches use code coverage as a heuristic to guide their path exploration efforts.

When it comes to generating syntactically valid inputs without spending too much computational resources on symbolic execution and constraint solving, one technique that is often used by black-box testing to surpass simple syntactic validity checks is mutational fuzzing. In contrast to purely *generational* approaches that create their test inputs completely from scratch, *mutational* approaches take existing input samples and change them in different ways to create new inputs. By changing provided inputs only slightly, these fuzzers are able to gradually produce inputs that are different enough to exercise new program behavior, yet similar enough to still be syntactically valid.

One such mutational fuzzer is American Fuzzy Lop (AFL) [13]. AFL is not a purely black-box approach either: it uses instrumentation or emulation to analyze the execution paths taken by the program under test. This allows it to produce inputs of complicated formats such as PNG or JPEG and successfully find bugs in corresponding libraries. One drawback of purely mutational approaches, however, is their dependence on the quality of the provided sample inputs, which may sometimes not be readily available, if at all. Especially with regards to efficiency, having to start from an empty input is a considerable disadvantage. For example, starting from an empty file, AFL took six hours on an eight-core machine to find a valid JPEG file[1].

[1]https://lcamtuf.blogspot.de/2014/11/pulling-jpegs-out-of-thin-air.html

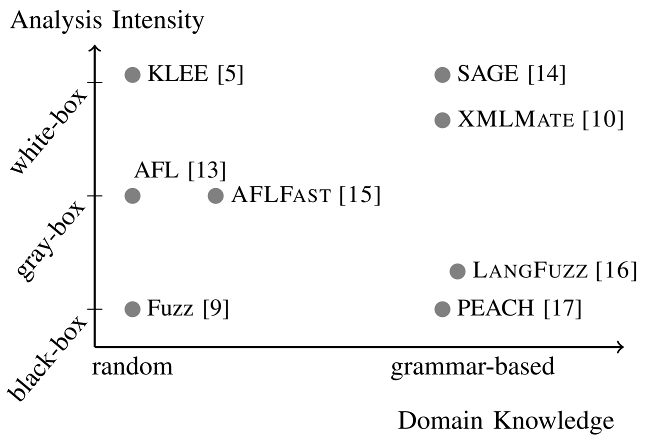

Figure 2. Test Input Generators

In contrast, the fuzzer PEACH [17] combines the generational and mutational approaches. It relies on a user-provided grammar describing the targeted input format to be able to produce syntactically valid inputs, which it later mutates. Some of its mutation operators aim to preserve the adherence of the inputs to the grammar, while others specifically introduce violations. Being more of a black-box fuzzer, PEACH forgoes any analysis of the program under test and simply generates lots of inputs to be executed against the program under test.

Holler et al. also successfully use a grammar-based black-box approach that is both mutational and generational [16] to find numerous defects in the JavaScript interpreters of the Firefox and Chrome web browsers as well as the PHP language interpreter. To achieve good coverage of the interpreter under test, they introduce additional post processing measures to make the generated inputs at least partially semantically valid.

Godefroid et al. go on to enhance their work SAGE with a grammar to achieve higher code coverage and exploring deeper program paths while avoiding dead-ends [14].

My own previous work on combining a grammar-based generational and mutational approach with a code coverage driven feedback loop by means of an evolutionary algorithm [10] was effective for the targeted subset of XML-based formats. Providing an interface for conversion of the resulting XML representations enabled a successful extension of this approach to arbitrary formats.

Böhme et al. created a path-coverage-based *gray-box* approach AFLFast [15] as an improvement to AFL. They leverage a Markov chain model of a partitioning of the input space to increase the efficiency of fuzzing by an order of magnitude.

III. Approach

The proposed work consists of three distinct parts. First, a fuzzer supporting interchangeable specifications and metrics is required as a common framework for experiments. Next, the analyses and metrics themselves have to be formalized and implemented. Beside the focus on grammar-based metrics, it is worth considering other readily available information sources like e.g. hardware counters in the pursuit of efficiency. Finally,

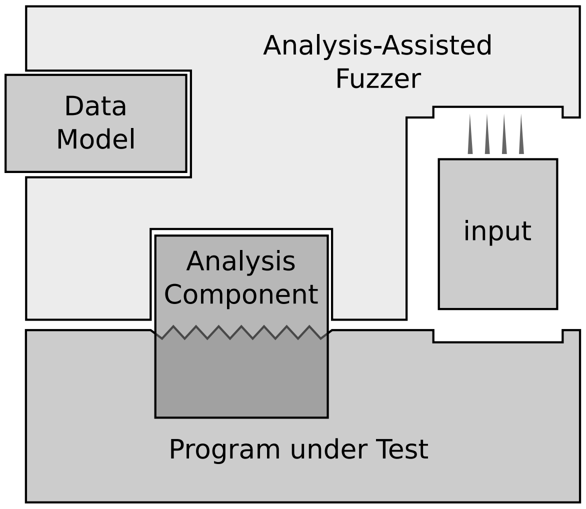

Figure 3. Analysis-Assisted Fuzzing Framework

a categorization of test subjects is necessary to be able to correctly gauge and generalize the experimental results.

A. Analysis-Assisted Fuzzing Framework

As a base for experimental investigation a unified fuzzing framework is required which allows for interchangeable analysis components as well as some form of input format specification. I propose to use an evolutionary algorithm [18] as the main driving loop of the fuzzer because both the data model (i.e. the format specification) as well as the fitness function (i.e. the analysis) can be interchangeably plugged into such an algorithm by design. Figure 3 shows the proposed fuzzing framework setup and its constituent components: the fuzzer is instantiated with two inputs: the data model and the analysis component. The data model represents the domain knowledge and most likely takes the form of a grammar specifying the syntactic rules of the input format. The analysis component provides the evolutionary algorithm with feedback on the quality of the inputs, such that it can make an appropriate selection for creating further inputs. The specifics of the analysis are completely encapsulated within the component, such that the fuzzer need only be concerned with the reported quality of the inputs. The analysis itself can be arbitrarily complex and invasive into the program under test or the data model.

B. Analysis Techniques

At the core of this work are the different lightweight analyses that can profitably enhance the effectiveness of a fuzzer. Depending on their main data source the analyses can be roughly divided into two categories: *model-based* and *subject-based*. Model-based analyses are applied to the generated inputs and their corresponding data model, while subject-based analyses mostly probe the application under test. The proposed initial set of analyses to implement includes the following:

- *Production coverage* is a model-based analysis which reports the fraction of production rules of the input grammar used in the creation of an input. By favoring inputs that cover more production rules, a more varied behavior is expected to be evoked in the program under test, thus leading to more defects revealed. This analysis seems most fitting for generating entire suites of inputs simultaneously because one input can hardly use all possible productions of a grammar at the same time. Fraser and Arcuri [19] show whole test suite generation to be effective for test cases.
- Another model-based analysis is *interval bounds analysis* which identifies intervals in the target space of grammar rules and aims to favor inputs that contain values close to the intervals' boundaries. This analysis aims at finding possible generalized forms of the off-by-one error in the program under test.
- The *validity distance* analysis aims to find inputs that violate the rules of the grammar while still being accepted by the program under test. This analysis is both model-based as well as subject-based in nature.
- *Cold path coverage* is a subject-based analysis which associates the execution path taken by the application with the input it is processing. This data enables gradual creation of inputs that cover paths that are only rarely executed.
- An indirectly subject-based analysis is *page miss frequency*, which relies on hardware support for efficiency and can be used to guide the generation of inputs towards stimulation of widely dispersed memory accesses.

These analyses are expected to enable lightweight acquisition of metrics to be used in the calculation of fitness values for inputs in the feedback loop of the fuzzer.

C. Program Categorization

The subject-based analyses have different costs and benefits that are dependent on some qualities of the programs they are applied to. Programs like compilers and interpreters are very challenging targets for such techniques as symbolic execution because of their sheer number of execution paths, and should thus profit more from model-based analyses instead. More streamlined programs, which are usually parts of data processing pipelines, on the other hand, may be better suited for use with subject-based analyses.

Consequently, finding the right analysis for a specific application under test creates the additional challenge of classifying the application according to some criteria to determine the suitability of the analysis in question.

Creating different criteria to enable this classification of programs is another task I aim to complete in my thesis. Intuitively, it seems promising to focus mostly on approximations of program complexity. *Cyclomatic complexity* [20] – a classic complexity measure of a program proposed by McCabe in 1976 – seems to be a good starting point. However, some programming language constructs like `switch` statements often tend to artificially increase the cyclomatic complexity value without actually increasing the real complexity of the program as perceived by a particular analysis technique. To counter this effect I propose to use a *modified cyclomatic complexity* that accounts for such phenomena. The *average method length* as well as the *average number of method calls* may prove to be successful complexity estimators as well.

IV. Evaluation

This work mainly focuses on the efficiency of automated grammar-based test input generators as compared to traditional random testing approaches. Therefore, a naïve random fuzzer shall be used in the evaluation as a baseline. In the process of evaluation of an analysis, it shall be used as the guiding analysis component in the fuzzing framework proposed in section III-A.

To determine the *effectiveness* and *efficiency* of grammar-based fuzzing compared to traditional coverage-based approaches (**RQ1**) the best analysis for each program category shall be compared against a coverage-based fuzz tester created with the fuzzing framework by using branch-coverage as the guiding analysis.

For measuring the effectiveness and efficiency of grammar-based fuzzing compared to random testing (**RQ2**) a comparison shall be performed against a fuzzer created with the framework by omitting the analysis component as well as the input grammar.

To find the analysis best suited for finding defects (**RQ3**), several test subjects from different program categories shall be fuzzed with each analysis.

To further classify and compare the quality of the proposed analyses, properties like *average time to defect* and *recall* (i.e. fraction of defects found) can be considered for some corpus of programs with known bugs. As a side note, the *precision* (i.e. the fraction of true positives) of any system-level fuzzer is `one` by construction because the program under test is exposed to its users and must handle all inputs gracefully, thus making every triggered defect a true positive.

V. Expected Contributions

In this work I propose to focus on raising the efficiency of state-of-the-art automated test input generation tools because they seem to be lacking when applied to real world subjects. I propose to heavily leverage automatically extracted context-free grammars describing the required input format, which has two major advantages:

Automatic extraction removes the need for manually documenting the format, which is not only laborious, but usually also error-prone. Additionally, once a grammar is extracted from one program it can be arbitrarily reused for any program that processes the same input format. Furthermore, grammars can be extended with additional information gathered from static analysis across multiple applications under test.

One problem of white-box approaches is their sacrifice of efficiency for effectiveness – the analyses they employ for guiding the test input generation are usually slow. I suggest to address this problem by making use of more lightweight analyses instead. These analyses are to be chosen in correspondence with the nature of the program under test. I further propose to define and evaluate criteria that can be used to categorize programs in terms of the type of lightweight analysis best suited for revealing defects.

Finally, I present a framework for analysis-assisted fuzz testing, designed to work with replaceable analysis components and data models. This framework is to be used in the creation of an efficient and practical automated test input generator.

Acknowledgment

This research has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7) / ERC Grant Agreement n. 290914 SPECMATE.

References

[1] M. Böhme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, April 2016.

[2] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" 2015.

[3] P. d. H. Nikolai Tillmann, "Pex - white box test generation for .net," in *Proc. of Tests and Proofs (TAP'08)*, vol. 4966. Prato, Italy: Springer Verlag, April 2008, p. 134–153.

[4] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *CAV*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 419–423.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 209–224, 2008.

[6] P. Godefroid, M. Y. Levin, and D. a. Molnar, "Automated Whitebox Fuzz Testing," *Search*, vol. 9, no. July, p. pdf, 2008.

[7] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. ACM, 2016, pp. 720–725.

[8] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[9] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[10] N. Havrikov, M. Höschele, J. P. Galeotti, and A. Zeller, "XMLMate: Evolutionary XML test generation," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-Nove, 2014. [Online]. Available: https://www.st.cs.uni-saarland.de/testing/xmlmate/

[11] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 370–379.

[12] N. Havrikov, "Search-Based Fuzzing of Binaries," Master's thesis, Saarland University, 2015.

[13] American fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[14] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *ACM SIGPLAN Notices*, vol. 43, no. 6, p. 206, 2008.

[15] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, pp. 1–12, 2016.

[16] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," *Usenix*, p. 38, 2012.

[17] Peach fuzzer: Discover unknown vulnerabilities. [Online]. Available: http://www.peachfuzzer.com/

[18] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Lulu, 2013, available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[19] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276 –291, feb. 2013.

[20] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.