Model-Based Fuzzing Using Symbolic Transition Systems — work in progress —

Wouter Bohlken
University of Amsterdam
Amsterdam, Netherlands
info@wouterbohlken.nl

Machiel van der Bijl
Axini B.V.
Amsterdam, Netherlands
vdbijl@axini.com

Ana Maria Oprescu
University of Amsterdam
Amsterdam, Netherlands
A.M.Oprescu@uva.nl

Graduate School of Informatics, University of Amsterdam

Abstract

As software is getting more complex, the need for thorough testing increases at the same rate. Model-Based Testing (MBT) is a technique for thorough functional testing. However, MBT cannot perform non-functional security testing. Fuzzing is a technique for automatically detecting vulnerabilities in software. Many different fuzzing approaches have been developed in the last years, ranging from random black-box to grammar-based white-box with structured input. In this research, we conduct a systematic review of state-of-the-art fuzzers and perform experiments where we combine multiple fuzzing approaches with MBT. Ultimately, we will choose the fuzzer that performs best, and integrate it into an MBT toolset.

We reviewed state-of-the-art fuzzing techniques and implementations and composed a list of candidate fuzzers that can be used in combination with MBT. We developed a generic wrapper that enables a model-based System Under Test (SUT) to be fuzzed with American Fuzzy Lop (AFL), a popular general-purpose fuzzer. Additionally, we developed a dictionary generator, that extracts basic model information and supplies it to AFL.

In: Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution, Virtual conference (originally in Amsterdam, the Netherlands), 01-02 July 2020, published at http://ceur-ws.org

1 Introduction

As software is getting more complex, thoroughly testing can be a time-intense and expensive task [Tre08]. Therefore, automating functional test generation is of high relevance. Model-Based Testing (MBT) is an up and coming technique for automating test generation, based on formally defined models [ULB+15]. Models can be expressed in Symbolic Transition Systems (STS) [FTW06], which allows the specification of data on transitions in the model.

While MBT is well-suited for functional testing, non-functional security testing is not covered in this approach. Therefore, Model-Based Security Testing (MBST) is introduced [SGS12], where model information is used to enable automated security testing. One technique that can be used in MBST is fuzzing. Fuzzing is a popular automated security testing approach where random, invalid, and/or unexpected input is supplied to the System Under Test (SUT) [TDM08]. The SUT is then monitored for crashes, memory leaks and/or inconsistencies. It is an effective technique for automatically finding security vulnerabilities and combines random testing with fault-based testing [FZB+16]. There are many different types of fuzzers, which can vary greatly on their program awareness, input structure awareness, instrumentation, intended purpose, and algorithms used [CCM+19].

While there are many fuzzers available, not all fuzzers are well suited to combine with MBT. For instance, when a fuzzer does not support some sort of input structure definition, there will be no way to supply it with model information. In this project, we will classify different state-of-the-art fuzzers, investigate how we can combine fuzzing with MBT, and to what extent we can use information from a model to enable more powerful fuzzing techniques. Using this information, one or more existing fuzzers will be integrated

into an MBT toolset. Efficiency and effectiveness are used as evidence measurements and commonly used in fuzzing research [FZB+16]. These evidence measurements qualify and quantify industrial applicability.

While MBST is considered a mature field [BCD+19], and model-based fuzzing has high relevance for industrial applications [SGS12, FZB+16], little research in the past years yielded production-level implementations of a model-based fuzzer. Moreover, no research has been done on MBT with LTS or STS in combination with fuzzing.

A proof of concept will be implemented using resources from Axini B.V., an Amsterdam based company that is specialized in MBT, more specifically, STS.

The following research questions are constructed.

RQ1: What is the most effective and efficient approach to integrate fuzz testing with STS?

- **RQ1.1:** How much more efficient and effective can a model-based fuzzer be, compared to a random fuzzer?
- **RQ1.2:** How can model information be used to define a grammar for generation-based fuzzers?
- **RQ1.3:** Are generation-based fuzzers more effective and efficient than mutation-based fuzzers when using STS?

RQ2: Which model information can be supplied to the fuzzer and what are the limitations?

RQ3: How many person hours can be saved by combining STS with fuzzing?

1.1 Contributions

Our research makes the following contributions:

1. **Comparative analysis of fuzzers**: in the form of a matrix, containing their characteristics.
2. **Benchmarks of different approaches**: containing efficiency, effectiveness, and coverage metrics.
3. **Implementation**: an integration of one or more fuzzers into the Axini Modeling Suite (AMS).

2 Background

In this section, we explain the characteristics of fuzzers, and the different techniques that have been proposed over the last years. Then we explain underlying techniques used for analysis and input generation, and explain the differences.

Fuzzer types can be divided into categories based on the following characteristics [TDM08, CCM+19]:

- **Smart or dumb:** this depends on whether or not the fuzzer knows about the input structure.
- **White-box, grey-box or, black-box:** this depends on whether or not the fuzzer knows about the program structure and performs analysis.
- **Mutation-based or generation-based:** where mutation-based takes a valid input and manipulates the content, and generation-based uses a given grammar to generate input.

Fuzzers are considered dumb when they are unaware of the SUTs input structure, and generate their seeds randomly, or by using a specific set of mutation algorithms. Smart fuzzers have some knowledge about the input structure, this can be a grammar or a set of keywords and possible values.

Fuzzers are considered white-box when they analyze the source code, this has the advantage that relevant tests are easily generated, and the program can be monitored for code coverage [TDM08]. The disadvantage is that monitoring and analysis can be time-consuming. Black-box fuzzing is used on compiled versions of software, where analysis is not performed by the fuzzer. This approach is faster but can also generate more irrelevant tests. Moreover, black-box fuzzers only scratch the surface of the application [CCM+19, PZK+17]. Grey-box fuzzing is defined as a blend of both techniques, typically a black-box is used, plus some run-time information to improve testing [TDM08]. In most cases, this includes code coverage metrics, to detect when a new branch is traversed, without performing static analysis.

Mutation-based fuzzers outperform generation-based fuzzers in terms of speed, because of its random nature, no generation has to be performed. However, a large amount of invalid input is mutated, this may cause the input being rejected early in the process, because it deviates too much from the defined structure. Therefore, mutation-based fuzzers almost always have lower code coverage than generation-based fuzzers [CCM+19]. Generation-based fuzzers, on the other hand, generate more valid input, but generation of relevant input data can be time-consuming [CCM+19].

Most industry-standard fuzzers are mutation-based grey-box, because mutation-based fuzzers greatly outperform generation-based fuzzers in terms of speed. Typically, smart algorithms like genetic algorithms are used to manipulate the valid input [CCM+19].

2.1 Dynamic analysis

Dynamic analysis is used to acquire information about the state of the SUT. This information is then used to decide what input to use in the next cycle. There

are mainly three types of dynamic analysis: coverage guided, dynamic symbolic execution and dynamic taint analysis.

2.1.1 Coverage-Guided Fuzzing

The most popular dynamic analysis technique is Coverage-Guided Fuzzing (CGF). CGF uses coverage metrics to detect when new interesting paths are reached. It uses random changes, deletions, or additions to a given input and retains interesting inputs. When a new path is discovered, the corresponding input is saved and used in concurrent tests [CCM+19]. This approach is very effective in reaching high coverage in a reasonable time-frame. However, when fuzzing programs that requires highly structured input, this approach suffers from the random nature of the input data and can get stuck at a certain coverage level.

2.1.2 Other techniques

Dynamic symbolic execution determines the possible inputs for the SUT by using symbolic values as inputs, and then constructing a path set. The main drawback of this technique is path explosion, which results in low efficiency, compared to CGF.

Similar to dynamic symbolic execution, dynamic taint analysis infers structural properties from input values. The main difference is that it keeps track of inputs that did not result in desirable paths, by tainting them.

2.2 Sample Generation

As mentioned before, fuzzers can either use a generation-based or a mutation-based approach to construct new inputs. There are two different strategies for mutation: random mutation and scheduling algorithms.

Random Mutation

The random mutation technique has no knowledge of the input structure, and only mutates given seeds. It has the advantage that it is easy to implement and highly scalable, since they do not rely on heavy monitoring or feedback. This approach is very ineffective in finding complex bugs, since it has no knowledge of program states and therefore does not know when a new part of the code is reached.

Scheduling algorithms

In contrast to random fuzzing, scheduling algorithms employ some optimization to maximize the outcome. The way these seeds are chosen depends on the fuzzer, and its goal. Scheduling algorithms can be combined with other algorithms such as simulated annealing and Markov, for further optimization.

Grammar representation

Grammar representation uses a grammar to constrain the input data structure [CCM+19]. This approach is most effective for fuzzing programs that require highly structured input, since it can reach a high coverage very quickly by using a grammar instead of completely random input. The drawback, however, is that it requires a grammar file that is typically written by hand, which is time-consuming and error-prone.

3 Review of State of the Art Fuzzers

We conducted a systematic review of state of the art fuzzers, we scanned Github repositories and searched the following digital libraries: IEEE, ACM, ScienceDirect, Springer, Wiley. We filtered on papers that were published in security proceedings since 2010, and used the following keywords: ("fuzz testing" **OR** "fuzzing"). Furthermore, we applied snowballing to find additional sources. Ultimately, we composed a list of fuzzers that can be combined with MBT. We filtered the list of fuzzers based on the following criteria:

- Production level, not a prototype or proof of concept
- Free to use, preferably open source
- Well maintained, the latest release should not be more than two years old
- Should support input structures
- Target programs should be general-purpose
- Should support Ruby

3.1 American Fuzzy Lop

American Fuzzy Lop (AFL) is the most popular state-of-the-art coverage guided fuzzer at the moment. AFL is considered dumb because it is not aware of the input structure, and only uses coverage metrics and genetic mutation algorithms to mutate given inputs, without using a grammar. AFL uses multiple scheduling algorithms to choose new seeds. Even though AFL is optimized for small inputs, it can be supplied with a dictionary. A dictionary is simply a list of keywords that can be used as inputs (e.g., for SQL database software this could include: INSERT, *, ;, %, etc). AFL uses this to construct new inputs, reducing its randomness and therefore, the amount of invalid input.

3.1.1 American Fuzzy Lop Extensions

The source code of AFL is publicly available and can be extended. Even though AFL is a general-purpose

fuzzer, it is not necessarily effective for all use cases. Therefore, in recent years, a substantial amount of research has been conducted, where AFL is used as a base and then extended to enhance performance in specific use cases. [KRC+18] These extensions add other algorithms to the sample generator, or completely replace it, while maintaining the other features.

AFLFast [BPN+16] uses a modified seed selection algorithm that is aimed to improve the efficiency of AFL. It uses power schedules to regulate the number of time spent on a single seed.

To overcome the inherent randomness of AFL's default implementation, a few extensions have been developed to increase the performance when dealing with programs that take more complex inputs.

Superion [WCW+17] adds the ability to supply a grammar to AFL, by using ANTLR [1]. A grammar file is converted to a parser, which is then used as a mutation strategy to choose the next input value. The mutation process consists of parsing the input into an abstract syntax tree, and then randomly manipulating its subtrees.

Driller [SGS16] is a supplementary tool for AFL that monitors the amount of time when AFL gets stuck, that is, not finding any new paths. When this occurs, Driller performs symbolic execution using angr [2] to find new valid inputs.

Similar to Driller, Skyfire [WCW+19] is a tool that can be used together with AFL. It focuses on programs that take highly structured inputs, and generates seeds that pass both syntax and semantic checks.

3.2 Other Coverage-Guided Fuzzers

Similar to AFL, Angora [CC18], libFuzzer [3] and honggfuzz [4] use the same CGF strategy with scheduling algorithms, but are less widely used and extended.

Nautilus [AFH+19] is a coverage-guided grammar-based fuzzer that focuses on efficiently fuzzing programs that require complex input structures. The authors state that most grammar-based fuzzers do not use coverage-guidance and therefore, lack the feedback on interesting paths, while most coverage-guided fuzzers use small mutations that lead to inefficiency when fuzzing programs with complex input structures.

4 Metrics

To define the optimal approach of the integration of a fuzzer into STS, we use the following metrics, proposed in [FZB+16]:

Effectiveness: in the context of this project, effectiveness can be measured by the number of faults found, possibly in relation to the amount of tests executed.
Efficiency: can be measured by relating artifacts such as faults and test cases to the time taken and costs.
Code coverage: quantifies the number of lines and methods that are traversed, this gives insight in the possibly hard to reach pieces of software that are not reached by the fuzzer.

4.1 Hypothesis

Using model information, we can implement a dictionary generator that extracts labels, and put this in a dictionary file that can be read by AFL (or other mutation-based fuzzers).

We hypothesize that when we generate a dictionary and feed this to a mutation-based fuzzer, the effectiveness will be increased, compared to fuzzing without a dictionary. Furthermore, we hypothesize that a grammar-based fuzzer can be useful when combined with MBT. However, we need to make sure that a grammar-based fuzzer will not only fuzz with valid inputs, but also generates test cases that range from completely invalid, to almost valid input. This way, we can cross the boundaries of the functional (and expected) behavior of the SUT.

5 Evaluation Setup

5.1 Benchmark

In an evaluation study on fuzz testing [KRC+18], 32 experiments have been evaluated and compared. This study shows that it is considered a common, and well-accepted practice to compare a new implementation against an AFL baseline. More specifically, it should use the same input seeds (since this can make a significant difference on the performance) and standard configuration parameters. For the benchmark, we use AFL and supply it with a dictionary that will be generated from a model, which converts labels into keywords. Additionally, we create a set of initial input seeds by hand, including a few paths through the model that are significantly different. The results of this test run will be compared to a baseline AFL run that is not supplied with a dictionary, but with the same input seeds. This way, we can quantify the efficiency, effectiveness, and code coverage of our approach, using the metrics stated in Chapter 4.

5.2 Proof of Concept

We developed an adapter to be able to communicate with SUTs. All the SUTs at our disposal are developed in Ruby, therefore, the adapter is also developed

[1] https://www.antlr.org
[2] https://angr.io
[3] http://llvm.org/docs/LibFuzzer.html
[4] https://github.com/google/honggfuzz

in Ruby, to maintain coverage guidance. To make AFL work with Ruby, an extension called Kisaten [5] is used. This extension enables instrumentation, coverage guidance and converts exceptions into AFL-detectable crashes. We created a generic AFL wrapper that connects to a given SUT over the command line, takes arguments, and runs them on the SUT. Every SUT that is modeled in AMS has the same interface, which means that, theoretically, any SUT can be fuzzed using this wrapper. To test the hypothesis in Sec. 4.1, we created a dictionary generator that parses all possible labels from a model and puts it in an AFL dictionary format file.

Unfortunately, fuzzing using the generated dictionary did not yet result in a significant improvement over fuzzing without a dictionary.

6 Discussion

The experiment guidelines proposed by [KRC+18] show that comparisons which run for less than 24 hours are not thorough enough, since results can significantly change within this time frame. Furthermore, because of the random nature of fuzzing, multiple runs with the same configurations can vary greatly, therefore, multiple runs have to be performed. Additionally, the size of the target corpus should be large, or at least a representative set. This is a big challenge and it is likely that not all of these guidelines can be followed properly, due to limited resources. However, ultimately, we will strive to deliver results that are statistically sound and significant.

Botella et al. claim that model-based fuzzing can only be used for robustness testing [BCD+19], and it is not possible to formalize security properties and provide evidence for the test strategy. However, this only means that security properties can not be defined in models, but the fuzzer will still be able to use its own set of security properties. Therefore, when implementing a model-based fuzzer, no security properties will be compromised.

6.1 Threats to Validity

The prototype of the model-based fuzzer will make use of the Axini Modeling Suite, which is not publicly available and thus, the benchmarks can not be replicated. However, Axini can be contacted for further details.

7 Related Work

Manès et al. conducted a survey [MHH+19] including a comparison, but used selection criteria that differ from ours (presented in Chapter 3). The purpose of their survey is to supply an overview of all fuzzing approaches and tools, presented in proceedings from January 2008 to February 2019.

Li et al. also conducted a survey [LZZ18], highlighting recent advances in fuzzing, which improves efficiency and effectiveness. This survey, however, only focuses on highlights in new techniques, including examples. These selection criteria do not focus on completeness and therefore do not cover all possible fuzzers that could be used in our research.

Schneider et al. [FZB+13] designed a model-based behavioral fuzzing approach where UML sequence diagrams are mutated by fuzzing operators. This results in invalid data which can then be fed to the SUT. In their approach, they eliminate overhead in the fuzzing process, by generating test cases at run-time, instead of generating all tests beforehand, and restarting the SUT after each test. Since this research, multiple fuzzers (including AFL) have implemented persistent-mode features, where *fork()* system calls are used to overcome the same unnecessary restart issue.

T-Fuzz [JSL+14] is a generation-based fuzzing framework for telecommunication protocols. It is integrated in a conformance testing environment, and extracts already existing model information which is then used for fuzzing. The results show that high code coverage can be reached, and once a new protocol including model is available, it can be fuzzed without much implementation effort.

8 Conclusion and Next Steps

We reviewed the recent developments regarding model-based fuzzing, fuzzing techniques, and implementations of fuzzers. Based on this knowledge, we composed a list of candidate fuzzers that will be investigated further. We created a generic wrapper for AFL, along with a dictionary generator. This enables SUTs that are modeled in AMS to be fuzzed with AFL, and automates the process of defining a dictionary, which can be a time-consuming process. These are the first steps towards a model-based fuzzer.

The next step is to repeat the same experiment with dictionaries on other mutation-based fuzzers, with a large test corpus. Additionally, we will use the MBT test case generator, extract interesting paths through the model, and convert these into input seeds. This process eliminates the manual work of creating input seeds and ensures a high model coverage. After that, we will implement a grammar generator and feed it to generation-based fuzzers. Ultimately we will compare the results and reach a verdict on the most effective approach.

[5] https://github.com/twistlock/kisaten

References

[AFH+19] Nautilus: Fishing for Deep Bugs with Grammars Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi and *Network and Distributed Systems Security (NDSS) Symposium 2019*

[BPN+16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. *Proceedings of the 2016 ACM SIGSAC Conference on computer and communications security, 24 October 2016*, pages 1032-1043. ACM, 2016

[BCD+19] Julien Botella, Jean-François Capuron, Frédéric Dadeau, Elizabeta Fourneret, Bruno Legeard and Florence Schadle. Complementary test selection criteria for model-based testing of security components. *International Journal on Software Tools for Technology Transfer*, pages 425-448.. Springer, 2019

[CC18] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. *39th IEEE Symposium on Security and Privacy* 2018. IEEE, 2018

[CCM+19] Chen Chen a, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo and Wenqian Liu. A systematic review of fuzzing techniques. *computers & security 75.* 2018, pages 118–137. Elsevier, 2018

[FTW06] L. Frantzen, J. Tretmans and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. *Formal Approaches to Software Testing and Runtime Verification*, pages 40-54. Springer Berlin Heidelberg, 2006.

[FZB+11] Felderer, Michael and Agreiter, Berthold and Zech, Philipp and Breu, Ruth. A Classification for Model-Based Security Testing. *VALID 2011 : The Third International Conference on Advances in System Testing and Validation Lifecycle.* IARIA, 2011

[FZB+16] Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler and Alexander Pretschner. Model-based security testing: a taxonomy and systematic classification. *Software Testing, Verification and Reliability volume 26.*, pages 119-148. Wiley, 2016

[JSL+14] William Johansson, Martin Svensson, Ulf E. Larson, Magnus Almgren, Vincenzo Gulisano. T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, March 2014, pp.323-332.* IEEE, 2014.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper Shiyi Wei and Michael Hicks. Evaluating Fuzz Testing. *Proceedings of the ACM Conference on Computer and Communications Security (CCS) 2018.* ACM, 2018.

[LZZ18] Jun Li, Bodong Zhao and Chao Zhang Fuzzing: a survey *Cybersecurity.* Springer, 2018

[MHH+19] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo "The Art, Science, and Engineering of Fuzzing: A Survey" arXiv, 2019

[PZK+17] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. CCS '17, Dallas, TX, USA. ACM, 2017.

[SGS12] Ina Schieferdecker, Juergen Grossmann and Martin Schneider. Model-Based Security Testing. Electronic Proceedings in Theoretical Computer Science, 2012.

[FZB+13] Martin Schneider, Ina Kathrin Schieferdecker, Jürgen Großmann, Andrej Pietschker. Online Model-Based Behavioral Fuzzing. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops.* IEEE, 2013

[SGS16] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. *Conference: Network and Distributed System Security Symposium*

[TDM08] Ari Takanen, Jared D. Demott and Charles Miller. Fuzzing for software security testing and quality assurance. *Artech House information security and privacy series*, 978-1-59693-214-2. Artech House, 2008.

[Tre08] Jan Tretmans. Model Based Testing with Labelled Transition Systems. Springer Berlin Heidelberg, 2008.

[ULB+15] Mark Utting, Bruno Legeard, Fabrice Bouquet, Elizabeta Fourneret, Fabien Peureux, Alexandre Vernotte. Recent Advances in Model-Based Testing. *Advances in Computers 00 (2015).* Elsevier, 2015.

[WCW+17] Junjie Wang, Bihuan Chen, Lei Wei and Yang Liu. Superion: Grammar-Aware Greybox Fuzzing. *IEEE Symposium on Security and Privacy.* IEEE, 2017

[WCW+19] Junjie Wang, Bihuan Chen, Lei Wei and Yang Liu. Skyfire: Data-Driven Seed Generation for Fuzzing arXiv, 2019