Model-Based Fuzzing Using Symbolic Transition Systems

Wouter Bohlken
info@wouterbohlken.nl

January 31, 2021, 48 pages

Academic supervisor: Dr. Ana-Maria Oprescu, a.m.oprescu@uva.nl
Daily supervisor: Dr. Machiel van der Bijl, vdbijl@axini.com
Host organisation: Axini, https://www.axini.com

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

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. Model-Based Testing uses a formal definition of a program and automatically extracts test cases. While MBT is useful for functional testing, non-functional security testing is not covered in this approach. Software vulnerabilities, when exploited, can lead to serious damage in industry. Finding flaws in software is a complicated, laborious, and expensive task, therefore, automated security testing is of high relevance. Fuzzing is one of the most popular and effective techniques for automatically detecting vulnerabilities in software. Many different fuzzing approaches have been developed in recent years. Research has shown that there is no single fuzzer that works best on all types of software, and different fuzzers should be used for different purposes.

In this research, we conducted a systematic review of state-of-the-art fuzzers and composed a list of candidate fuzzers that can be combined with MBT. We present two approaches on how to combine these two techniques: offline and online. An offline approach fully utilizes an existing fuzzer and automatically extracts relevant information from a model, which is then used for fuzzing. An online approach utilizes fuzz data generators and integrates them into an MBT toolset. We design and implement frameworks for both the offline and online approaches, which we call 'MOfuzz' and 'Fuzzion', respectively. We test our MOfuzz implementation with AFL, a popular coverage-guided grey-box fuzzer. The implementation of Fuzzion uses three general-purpose fuzzers, Radamsa, Peach, and Fuzzino. Our results indicate that the combination of fuzzing with MBT works for finding defects. Fuzzion offers a generalized and simple interface for integrating multiple fuzzers, eliminating the manual labor of configuring fuzzers for every SUT. Moreover, our results show that different fuzzers trigger different types of defects. Peach found a large number of defects on an integer argument, Radamsa only found defects in string argument, and Fuzzino mostly detected timeouts. This significant difference in the types of defects found indicates that it is beneficial to integrate multiple fuzzers.

Contents

Chapter 1

Introduction

As software is getting more complex, thorough testing can be a time-intensive and expensive task [1]. Therefore, automating functional test generation is of high relevance. Model-Based Testing (MBT) [1] is an up and coming technique for automated test generation, based on formally defined models. The growth of online applications and services increases the number of potential security vulnerabilities [2]. In recent years, vulnerabilities have become a serious problem in industry and can result in significant damage [3–5]. One of the most popular and effective and efficient techniques for automated security testing is fuzzing, due to its simplicity, scalability and large amount of evidence for finding real-world bugs [3, 4, 6–9].

Fuzzing is an automated security testing technique, where random, invalid, and/or unexpected input is supplied to the System Under Test (SUT) [6, 10, 11]. The SUT is then monitored for crashes, timeouts, memory leaks, and/or inconsistencies. It is an effective technique for automatically finding security vulnerabilities and combines random testing with fault-based testing [5]. The goal of fuzzing is to crash a system and to expose potential vulnerabilities. Many fuzzing tools have been developed in the last decade, with many different techniques and target programs. Many fuzzers target a specific type of software, such as OS kernels, binaries or web servers [6, 7]. In contrast, there also exist general-purpose fuzzers that aim to use general heuristics on any type of software. Research has shown that there is no single fuzzer that performs best on any type of software [8, 10, 12]. Different SUTs require different types of strategies to be the most effective. When comparing fuzzer's performances, the number of bugs found can vary greatly per SUT. Furthermore, fuzzers use different strategies that might yield different type of bugs [7, 8, 13]. Therefore, it is beneficial to use multiple fuzzers to find defects [10].

1.1 Problem Statement

Model-Based Testing is well-suited for functional testing, but non-functional security testing is not covered in this approach. Therefore, Model-Based Security Testing (MBST) is introduced [14], where model information is used to enable automated security testing. While Mobel-Based Security Testing is considered a mature field [13], and model-based fuzzing has high relevance for industrial applications [5, 14], little research in the past years yielded production-level implementations of a model-based fuzzer. Moreover, none of the production-level implementations is general purpose. MBT allows for modeling many different types of software, therefore, utilizing multiple fuzzers is beneficial to fuzz each target effectively.

Our goal is to combine the formalism of models with state of the art fuzzing techniques, using a generic framework. Ultimately, we aim to combine these two techniques to systematically fuzz modeled SUTs, and eliminate the manual labor of implementing specific fuzzers for each SUT. In this project, we reviewed state-of-the-art fuzzing techniques, designed two frameworks for combining existing fuzzers with MBT, and developed a prototype that implements these frameworks.

1.1.1 Effectiveness and Efficiency

In fuzzing research, *effectiveness* and *efficiency* are the most commonly use metrics to quantify and qualify the improvements over state of the art [4, 8, 15]. Most of these papers, however, don't supply a clear definition of these terms. Therefore, we use the definitions proposed by Felderer et al. [5]:

Effectiveness: typically measured by the number of faults found, possibly in relation to the number

of tests executed.

Efficiency: can be measured by relating artifacts such as faults and test cases to the time taken and costs.

1.1.2 Research questions

Based on the problem statement, the following research questions are constructed.

RQ1: How can we effectively and efficiently integrate fuzz testing with Symbolic Transition Systems?

RQ1.1: To what extent does a smart model-based fuzzer outperform a dumb fuzzer?

RQ1.2: How can model information be used to define input specifications for generation-based fuzzers?

RQ1.3: How does the performance of generation-based fuzzing differ from mutation-based fuzzing, in terms of effectiveness and efficiency?

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

RQ3: Which manual tasks can be reduced or eliminated by combining Symbolic Transition Systems with fuzzing?

1.1.3 Research method

For this project, exploratory research was performed, where multiple fuzzers were reviewed and compared in terms of effectiveness and efficiency. The results of this research include a detailed comparison for all reviewed tools, their strengths and weaknesses (limitations), awareness, and algorithms used.

1.2 Contributions

Our research makes the following contributions:

1. **Comparative analysis of fuzzing approaches and tools**, including a methodology to decide which approaches can be combined with MBT, and how.
2. **Model-Based Fuzzing Frameworks**: designs of two frameworks for combining fuzzers with MBT
3. **Model-Based Fuzzing Frameworks Proof of concept**: an integration of multiple fuzzers with an MBT toolset.
4. **Experiments using the proof of concept**: Multiple runs of our proof of concept on real-world software.

1.3 Project Context

This project was executed in corporation with Axini, an Amsterdam-based company that specializes in MBT. Their MBT toolset is called Axini Modeling Platform (AMP).

1.4 Outline

In Chapter 2 we describe the background of software testing, model-based testing, security testing, and fuzzing. Chapter 3 describes the review of state-of-the-art fuzzing approaches and tools. Chapter 4 describes our frameworks for integrating fuzzing with MBT and our implementations. Chapter 5 describes the experimental setup for our implementations. The experiment results are presented in Chapter 6. Chapter 7 discusses the implementation, benchmark and results. Chapter 8 contains the work related to our systematic review and model-based fuzzing. Finally, we present our concluding remarks in Chapter 9 together with future work.

Chapter 2

Background

2.1 Software Testing

Software testing is the process of verifying that the SUT works as it is specified. This process can take many forms and can be done by hand or automatically. Testing software can expose defects, and/or gain confidence in the absence of defects. In software testing, either manual or automatic, one supplies test cases to the SUT, and then checks whether the output matches a specification. Ideally, one should test a system for every possible input in every possible code path. In most software systems, this is impossible to achieve. To deal with this restriction, heuristics are typically used, to test a system using a representative test set that covers all major paths.

Software testing can be performed at any stage of the development process. In agile software development, the development process typically includes *continuous integration*, where every code change is tested against a set of test cases. Testing can be performed on multiple abstraction levels. *Unit testing* tests the functionality of individual components. *Integration testing* tests whether different components interact correctly. *System testing* tests the entire software system and finally, *acceptance testing* can be performed to verify customer's requirements. In this section, we describe different aspects and approaches of software testing, relevant to this research.

2.1.1 Functional and Non-functional Testing

Functional testing covers the verification of functional properties of a program. These properties can be defined in a requirements document and describe how features should work. Non-functional testing, on the other hand, tests for requirements that are not related to specific features. This can include requirements e.g.: performance, scalability, or security.

2.1.2 White-box, Black-box and Grey-box Testing

The "box" definition describes the view on a SUT when writing tests. In white-box testing, the internal structures of a program are tested, with access to the source code. This approach can be used for unit, integration, and system testing. In this approach, metrics such as code coverage can be extracted. In a black-box approach, one does not use the internal source code to write test cases, but rather tests the SUT for input-output conformance, based on a specification. Black-box testing verifies that a system behaves as it should, and not *how* it works. Grey-box testing is something in between, where some knowledge of the internal workings of the system is used at a black-box level. This knowledge can be deduced when a tester has access to the source code, or by using reverse engineering and can include properties such as boundary values. Grey-box testing can also include instrumentation for coverage feedback, without performing program analysis.

2.2 Model-Based Testing

Many testing approaches used in industry rely on manually writing tests. Developers have to come up with relevant test cases and implement these. This is a time-consuming task and lacks systematic automation. When the specification of a system changes, the tests related to change, should be updated as well. The goal of MBT is to overcome this manual labor, by defining a formal model, which is then

used to derive relevant test cases automatically. When the specification of a system changes, only the model has to be changed. In the MBT process, inputs are generated and executed on the SUT, and the expected (modeled) output is checked against the actual output. MBT is a form of *black-box functional* testing, because the SUT is only monitored on input-output conformance and no program analysis is performed. Models only include the functional behavior of the SUT, and do not include non-functional properties.

Studies show that MBT detects significantly more faults than manual testing, but initially requires more activities, such as modeling and setting up a test infrastructure [15]. To let an MBT toolset communicate with the SUT, an adapter has to be implemented. An adapter is responsible for executing generated test cases on the SUT and report the response back to the MBT toolset. Since every SUT is different, there is no way to define a single adapter for all SUTs. Instead, adapters implement a generic interface. Using this interface, an MBT toolset can execute tests on any modeled SUT, without the need for specific implementations.

2.2.1 Transition Systems

Model formalisms can be expressed in the form of a transition system, where the transitions represent possible actions that can be performed to reach a different state. In a *labelled transition system* [1] (LTS), transitions are complemented with labels. Labels define observable actions of a system and represent the interaction of the SUT with its environment. In an LTS, it is impossible to define data values and variables. In large systems, this inevitably leads to state space explosion, where consecutive combinations of values lead to different transitions and states. LTS can be extended in the form of a *symbolic transition system* [16] (STS). STS allows for defining data values and variables. Transitions can be dependent on a certain condition for these values. This addition solves the state space explosion problem, because different states can be grouped, where variables and data values determine the differences. An STS is somewhat similar to a *state machine*, defined in UML.

2.3 Security Testing

Security testing can be grouped into two principal approaches: *security functional testing* and *security vulnerability testing* [5, 15]. Security functional testing is more or less a form of classical functional testing, but tests if security requirements are implemented correctly. For instance: "After 3 failed login attempts, a user account is blocked.". In security vulnerability testing, a system is tested for unintended behavior, due to flaws in a system design, implementation or configuration. This form of testing requires specific knowledge about security properties, and is hard to automate [5]. Specific types of attacks that expose vulnerabilities have to be defined and evaluated. Developers often lack this kind of knowledge, therefore, specialized companies can be used to perform these kinds of tests.

Application security testing can be grouped into *static application security testing* (SAST) and *dynamic application security testing* (DAST) [15, 17]. SAST typically performs static code analysis, to find known vulnerabilities. DAST dynamically checks for vulnerabilities, during execution of the SUT.

2.4 Fuzzing

Fuzzing is a form of security vulnerability testing, where security heuristics and attack patterns can be used to expose vulnerabilities in a system. Traditionally, fuzzing was a form of random black-box testing. A fuzzer did not perform analysis of the program and simply supplied random inputs to crash the SUT, or trigger unintended behavior. Since the introduction of fuzzing, many different and more sophisticated approaches have been developed. Nowadays, a wide variety of fuzzers can be found. Fuzzer types can be divided into the following categories [6, 18]:

- **Smart** or **bumb**: this depends on whether or not the fuzzer has knowledge 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. This reflects the definitions for software testing in general.
- **Mutation-based** or **Generation-based**: where mutation-based takes a valid input and manipulates the content, and generation-based construct inputs from scratch, typically using a grammar

or other forms of input definitions.

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 at least some knowledge about the input structure, this can be a grammar or input type and boundary definition [3, 10, 19].

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 [10]. 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 [6, 20]. 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 [10]. In most cases, this includes code coverage metrics, to detect when a new branch is traversed, without performing static analysis.

Mutation-based fuzzers typically outperform generation-based fuzzers in terms of speed, because of their 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 [6]. Generation-based fuzzers, on the other hand, generate more valid input, but the generation of relevant input data can be time-consuming [6].

2.4.1 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 three main types of dynamic analysis: coverage guided, dynamic symbolic execution, and dynamic taint analysis.

Coverage Guided Fuzzing

The most popular dynamic analysis technique is Coverage Guided Fuzzing (CGF) [4]. CGF uses coverage metrics to detect when new interesting paths are reached. It typically uses random changes, deletions, or additions to valid sample inputs and retains interesting inputs for further mutation. When a new path is discovered, the corresponding input is saved and used in concurrent tests [6, 19]. This approach is very effective in reaching high coverage in a reasonable time-frame. This is because it does not perform heavy analysis, but instrumentation to discover new code paths. The combination of genetic algorithms and feedback-driven fuzzing mode, allows for high flexibility, as well as customizability. However, when fuzzing programs that require highly structured inputs, this approach suffers from the random nature of the input data and can get stuck at a certain coverage level.

Some coverage guided fuzzers support the use of dictionaries. A dictionary is a list of keywords for the SUT, which can be used to construct new inputs, reducing its randomness and therefore, the amount of invalid input. For instance, in SQL database software, these could include: INSERT, *, ;, SELECT, MAX, etc).

Dynamic Symbolic Execution

Dynamic symbolic execution determines the possible inputs for the SUT by using symbolic values as inputs, and then constructs a set of possible paths. The main drawback of this technique is path explosion, which results in low efficiency, compared to coverage-guided fuzzing. Therefore, dynamic symbolic execution is not widely used.

Dynamic Taint Analysis

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. This technique can be combined with dynamic symbolic execution and random mutation. The main drawback of this method is that values can easily get tainted when they should not be, or the other way around.

2.4.2 Sample Generation Techniques

Sample generation is the process of generating or mutating samples that are used for fuzzing, and contains input restrictions. There are three main techniques for sample generation: random mutation, scheduling algorithms, and grammar representation [6].

Random Mutation

The random mutation technique has no knowledge of the input structure and only mutates given samples. It has the advantage that it is easy to implement and highly scalable, since it does 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. This approach is most effective for fuzzing programs that require highly structured inputs, since it can reach a high coverage very quickly by using a grammar, instead of being dependent on a random mutation triggering a new path. The drawback, however, is that it requires a grammar file that is typically written by hand, which is time-consuming and error-prone. Generation-based fuzzers always use a formal input specification and generally achieve higher code coverage than mutation-based fuzzers [6].

2.4.3 Fuzz Data Generators

Fuzzers are typically developed to perform the complete process of fuzzing, from sample generation or mutation, to execution on the SUT, to detecting crashes. In some cases, however, it can be beneficial to split this process over multiple tools and integrate them. For instance, when a fuzzer has to be integrated into a preexisting test environment. In such an environment, the communication and monitoring of the SUT would already be configured and, in this case, only fuzz data is needed to perform fuzzing runs. This is where fuzz data generators come in. They are not full-fledged fuzzers, but only generate data, based on a given input specification. The generated data is then typically written to a console, or an output file, which can then be read an executed by the testing environment. Some full-fledged fuzzers also support this feature, by specifying the target in such a way that it writes the inputs to a file.

2.5 Model-Based Security Testing

Model-Based Security Testing (MBST) refers to the process of testing security requirements in a model-based way [5]. MBST is suitable for DAST, and can be used for both functional and vulnerability testing. MBST is commonly used for systematically specifying and documenting security test objectives and automated test generation.

2.5.1 Model-Based Fuzzing

Like MBST, Model-Based Fuzzing can also be used for both functional (MBFST) and vulnerability testing (MBVT). MBFST is often referred to as *behavioral fuzzing* [14, 15]. In behavioral fuzzing, the SUT is fuzzed and then monitored to determine if it still behaves the way it is specified. MBFST has been proven effective for testing security properties. However, many security properties are not specifically stated in documentation, or modeled, this is where MBVT comes in. In short, MBVT is about "verifying that users cannot use a system in a way it was not intended to be used." [15]. Data generators in Model-Based Fuzzing can both use a generation-based, as a mutation-based approach.

2.6 Application Frameworks

An application framework is a reusable set of components that can be used to build custom applications [21]. The goal of a framework is to eliminate manual and error-prone labor that comes with developing applications and, therefore, reduce costs and risks. A framework typically contains core concepts and components that are generally used when developing applications. Since software complexity rises at a consistently steady pace, the need for generic implementations of these core concepts and components is of great importance.

Fayad and Schmidt [21] state that the benefits of object-oriented application frameworks include the following:

- **Modularity**: encapsulating volatile implementation details behind stable interfaces
- **Reusability**: definitions of generic components that can be applied to create new applications
- **Extensibility**: the ability to extend existing functionality by inheriting base classes and/or overriding methods.
- **Inversion of control**: the control flow being per the framework, rather than the caller. Event handlers can be customized by the application.

Chapter 3

Systematic Review of State of the Art Fuzzers

In this chapter, we explain our systematic review of state of the art fuzzers and compare the selected candidate fuzzers based on their features.

Section 3.1 explains our two different approaches to model-based fuzzing, including their corresponding requirements for candidate fuzzers. Section 3.2 explains how we searched for fuzzers. In Section 3.3 we discuss the selected fuzzers that we found and compare their features.

3.1 Online and Offline Approaches

When combining the concept of fuzzing with MBT, there are two possible approaches: **offline** and **online**. These two approaches reflect the way the test execution is performed on the SUT, this can either be done by a fuzzer, or an existing MBT toolset. We explain the architecture and implementation of these approaches in Chapter 4.

3.1.1 Offline Fuzzing

In an **offline** approach, the fuzzer is responsible for the entire process, but will make use of adapters to communicate with SUTs. Without an adapter, a separate implementation needs to be developed for each SUT that is fuzzed. Fuzzing the adapter makes our approach generic, because adapters always implement the same interface. For offline fuzzing, we use an existing model to extract information about labels, inputs and possibly generate valid sample values that can be mutated.

This approach is considered **offline**, because it does not utilize the features of the MBT toolset, such as generating inputs, reading responses, and traversing the model. In this approach, there is no model-coverage feedback. For this approach, we can utilize fuzzers that support some form of input structures.

3.1.2 Online Fuzzing

In an **online** approach, we use all the functionality of the MBT toolset, but use a fuzzer to generate (semi-)invalid inputs. In this case, we have full control over the existing functionality of the MBT toolset. This functionality can be used to traverse a model and let the SUT reach a certain state, before we start to fuzz. Furthermore, the behavior of the SUT can be monitored after executing a fuzzed value. In this case, we can guarantee full model-coverage, because all the traversed paths are monitored, and we can keep track of paths that have not been fuzzed yet. For this process, we can only use fuzzers that can generate data, without having to use all the other functionality.

3.2 Review Method

We conducted a systematic review of state of the art fuzzers, we scanned Github repositories and searched the following digital libraries: ACM, IEEE, 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 implementation, not a prototype or proof of concept
- Free to use, preferably open source
- Should not be deprecated
- Target programs should be general-purpose
- (Online only): Should support standalone data generation
- (Offline only): Should support inputs structure definitions

Using these selection criteria, we searched recent literature and Github repositories, and built a comparison matrix. See Table 3.1 for the full matrix.

3.3 Selected Fuzzers

In this section, we discuss the fuzzers we found in our survey. We grouped these fuzzers into standalone fuzzers, which can supervise the complete fuzzing process, and fuzz data generators, which can be used for integration into an MBT toolset. This distinction reflects the difference between the offline and online approaches. For each fuzzer, we discuss the key features, main characteristics, and, when applicable, the strategies and/or algorithms that it uses.

3.3.1 Standalone Fuzzers

In this section, we discuss the standalone fuzzers we found, using our selection criteria. All these fuzzers can be used for our offline approach, and some can also be used as data generators, which makes them suitable for our online approach as well.

American Fuzzy Lop

American Fuzzy Lop (AFL) is one of the most widely used coverage guided fuzzers at the moment [6–8, 22, 23]. AFL is grey-box and considered dumb because is it not aware of the input structure, and only uses coverage metrics and genetic mutation algorithms to mutate given inputs. AFL uses multiple scheduling algorithms to choose new seeds. Even though AFL is dumb, it can be supplied with a dictionary. The source code of AFL is publicly available and can be extended. AFL is widely used, because it is relatively easy to implement and proven effective.

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 done, where AFL is used as a base and then extended to enhance performance in specific use cases. As stated in Section 2.4.1, coverage-guided fuzzing allows for high effectiveness, flexibility, and customizability, which explains why there is a vast amount of AFL-related fuzzers. These extensions add other algorithms to the sample generator, or completely replace it, while maintaining the other features.

AFL++

AFL++ [1] is an extension on AFL that combines a set of improvements. It is not developed to serve a specific goal, but to increase performance and compatibility. One of it's key feature is the 'unicorn mode', which allows fuzzing binaries from different platforms, such as Android. Furthermore, features and patches developed by the community have been implemented, such as the power schedules implemented in AFLFast (see below), and mutation strategies from other fuzzers (Radamsa and honggfuzz).

AFLFast

AFLFast [24] uses a modified seed selection algorithm that is aimed to improve the efficiency of AFL. This seed selection algorithm is referred to as 'power schedules'. The key difference of AFL and AFLFast is that AFLFast assigns a higher probability to seeds that trigger paths that are not frequently exercised, and conversely, assigns less probability to seeds that trigger less frequently visited paths. The benchmarks

[1] https://github.com/AFLplusplus/AFLplusplus

performed at the time of writing (2016), show that AFLFast outperforms default AFL by an order of magnitude, however, recent developments in AFL yielded a significant performance boost as well. Furthermore, AFL++ includes the power schedules from this tool, on top of the most recent version of AFL.

AFLGo

AFLGo [25] extends AFL, to allow for directed fuzzing. The authors argue that existing grey-box fuzzers can not be directed into specific locations in a codebase. Directed fuzzing can be useful when the goal is to test a specifically vulnerable piece of code, a newly implemented patch, critical system calls, or a newly implemented piece of code. The authors implemented a simulated annealing algorithm that assigns more energy to seeds that are closer to the end goals and vice versa.

AFLSmart

AFLSmart [19] adds high-level input structure awareness to AFL, making it smart. This fuzzer is specifically designed to fuzz SUTs that take highly structured inputs. One can supply it with an input specification, which allows more control over format and strategies than the default AFL implementation. AFLSmart uses Peach (see below) to generate inputs, and apply these to AFL, combining the best of both worlds.

Angora

Similar to AFL, Angora [26] is a coverage-guided grey-box fuzzer. The key difference with AFL is that Angora uses taint tracking, which determines the exact bytes that trigger new paths. This allows the fuzzer to generate new inputs more efficiently. Angora and AFL are similar in their classification and target programs.

Driller

Driller [27] is a supplementary tool for AFL that monitors when AFL gets stuck for a certain amount of time, that is, not finding any new paths. When this occurs, Driller performs concolic execution using angr [2] to find new valid inputs. The authors argue that both fuzzing and concolic execution suffer from weaknesses, where fuzzing is incomplete and can not reach full code coverage and concolic execution suffers from path explosion.

FairFuzz

FairFuzz [28] is an AFL extension that changes the mutation algorithm of AFL to target rare code branches. This tool achieves this by counting the amount of times code paths are hit. The benchmarks performed with this tool indicate that code coverage is achieved faster on programs with many nested conditions, compared to default AFL.

The validity of this extension is criticized by Klees et. al. [8], because the authors only take code coverage into account, when comparing this tool against its default AFL counterpart. The benchmarks don't show that FairFuzz is more effective in finding bugs.

honggfuzz

honggfuzz [3] has the same characteristics as AFL. There are no significant differences.

libFuzzer

libFuzzer [4] has the same characteristics as AFL and honggfuzz. There are no significant differences.

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

Nautilus

Nautilus [23] is developed for fuzzing programs that take highly-structured inputs. It used grammar files, defined in Python structs, to generate random valid inputs. These inputs are then used by AFL as seeds. By default, Nautilus is used as a stand-alone fuzzer, but it can also be used in combination with AFL, to influence the next input seeds used by AFL. Furthermore, the input seed generator can be used separately. Therefore, Nautilus can be used for both approaches. The Nautilus grammar only consists of predefined values, rather than ranges. Therefore, in an online, stand-alone approach, it will not be useful for generating integer values that have min and max values. This functionality is already present in our MBT toolset and there, Nautilus would not be useful for integration.

Peach

Peach [5] is a smart black-box that supports both data mutation and generation. The input structure and target SUT are modeled in XML configuration files, called 'Pits'. Pits are divided into 2 categories, data pits and target pits. Data pits determine what kind of data is generated or mutated, whereas target pits describe how the SUT will be fuzzed. Pits can be configured in two separate files, to allow for reusability, or in a single file. Data pits support a wide range of configuration parameters, including the type of generators, input boundaries, and example seeds. Target pits can consist of command line parameters, paths to executables, server connections including host, port, path, and parameters.

The **Strategy** element determines the order of iterations. This can be set to an exhaustive deterministic sequence of outputs, or completely random and infinite. An additional **MaxFieldsToMutate** parameter determines how many values will be generated per strategy. See Listing 3.1 for an example pit and Listing 3.2 for example output values.

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://peachfuzzer.com/2012/Peach ../peach.xsd">
  <DataModel name="TheDataModel">
    <String value="HAL 9000" />
  </DataModel>

  <StateModel name="State" initialState="State1">
    <State name="State1">
      <Action type="output">
        <DataModel ref="TheDataModel"/>
      </Action>
    </State>
  </StateModel>

  <Test name="Default">
    <StateModel ref="State"/>
    <Publisher class="Console" />
  </Test>
</Peach>
```

Listing 3.1: Peach Pit Example

[5] https://www.peach.tech/products/peach-fuzzer/

```
[[ Peach v3.0
[[ Copyright (c) Michael Eddington

[*] Test ‘Default’ starting with random seed 5158.

[R1,-,-] Performing iteration
HAL 9000
[1,-,-] Performing iteration
[*] Fuzzing: TheDataModel.DataElement_0
[*] Mutator: StringMutator
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[R2,-,-] Performing iteration
HAL 9000
[2,-,-] Performing iteration
[*] Fuzzing: TheDataModel.DataElement_0
[*] Mutator: UnicodeBadUtf8MutatorÀÁ

 9000 HAL ø©≪ªüàø§Á9000[
```

Listing 3.2: Peach Output Example

SlowFuzz

SlowFuzz [20] aims to target program branches that have high complexity and slow down performance. The main goal of this fuzzer is to find code paths that trigger the worst-case behavior and therefore result in slow performance. These paths could be exploited in i.e. Denial-of-Service attacks. Rather than using coverage-guidance, SlowFuzz uses resource-usage-guided evolutionary search algorithms to find inputs that result in high resource utilization.

Superion

Similar to Nautilus, Superion [22] extends AFL and targets programs that take highly structured inputs. It adds the ability to supply a grammar, by using ANTLR [6]. A grammar file is converted to a parser, which is then used as a mutation strategy to choose the next input value. Superion introduces two new mutation strategies. The first is ‘Grammar-Aware Trimming’, which alters the trimming strategy of AFL. The default trimming strategy tries to reduce input seeds as much as possible, while it reaches the same code paths. This is effective for programs with lower structured inputs, but results in many invalid inputs when fuzzing highly structured programs, and can get stuck easily. Superion alters this strategy and ensures that trimmed inputs are always valid, according to the given grammar. The second strategy is ‘Enhanced Dictionary-Based Mutation’. Similar to ‘Grammar-Aware Trimming’, this strategy adds grammar awareness to the way dictionaries are used by AFL. The default dictionary mutation strategies result in destroyed structures and will therefore usually not reach deep code paths. This new strategy makes sure that dictionary mutations are valid, according to the grammar.

[6] https://www.antlr.org

3.3.2 Data Generators

In this section, we discuss data generators. These tools can only be used to generate input values and can not be used for the fuzzing process itself. These tools are suitable for an online model-based fuzzing implementation.

Fuzzino

Fuzzino [7] was developed by Fraunhofer FOKUS, a research and development group that specializes in security testing, as well as MBT. The authors state that Fuzzino is developed to be integrated into a testing tool, and mention MBT as an example. Fuzzino has knowledge of the input structure, when specified. The authors distilled multiple fuzzing heuristics, taken from other fuzzers such as Peach and Sulley [8], as well as other research on fuzzing. Rather than developing yet another fuzzer, the authors aim to provide a single tool that combines multiple approaches, strategies, and heuristics, while also reducing the number of test cases generated for an input.

Requests can consist out of one or more generators, and an optional specification. Generators can be used to generate data to serve specific purposes, such as SQL injections, cross-site scripting, bad unicode characters, and malformed XML. Specifications describe the input type and can be set to, e.g.: SQL, date, number, or IP address. Specifications can include parameters such as min value, max value, amount of bits, and the character encoding. Based on a specification, Fuzzino chooses and configures the appropriate generators.

Fuzzino is written in Java and can be integrated directly into projects that are also written in Java. For other languages, XML requests can be used. The number of outputs can be influenced by setting the **maxValues** on a Request. The output for requests is always deterministic, this can not be changed. To implement randomization, one can either fetch all fuzzed values for a request and then shuffle it, or set a random value for the **seed** parameter.

Listings 3.3 and 3.4 show an example of a basic number request. In this request, we specify that the minimum value is 0 and the maximum value is 42. Fuzzino uses a boundary value heuristic on numbers, and yields the following fuzzed values: 0, 42, 21, 14, 10, 5, 2, 1, -1, 43. Listings 3.5 and 3.6 show the request and corresponding response for an SQL injection. We omitted most of the output values, to save space, and instead listed some outputs from different types of SQL injections. The complete response contains many more variations of these outputs.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <ns2:request xmlns:ns2="http://fuzzino.fuzzing.fokus.fraunhofer.de/request">
3   <number name="Number" maxValues="500" seed="-1">
4     <specification minValue="0" maxValue="42" bits="32"/>
5   </number>
6 </ns2:request>
```

Listing 3.3: Fuzzino number request example

[7] https://github.com/fraunhoferfokus/Fuzzino
[8] https://github.com/jtpereyda/boofuzz

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:response xmlns:ns2="http://fuzzino.fuzzing.fokus.fraunhofer.de/response">
    <number id="a75e2886-8747-40c6-8cc8-bf1eb67041a8" moreValues="true" name="Number">
        <generatorBased>
            <generator name="BoundaryNumbers">
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:long">0</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:long">42</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                ...
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:long">-1</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:long">43</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
            </generator>
        </generatorBased>
    </number>
</ns2:response>
```

Listing 3.4: Fuzzino number response example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:request xmlns:ns2="http://fuzzino.fuzzing.fokus.fraunhofer.de/request">
  <string name="SQL">
    <generator value="SQLInjections"/>
    <generator value="SQLTimeBasedInjections"/>
  </string>
</ns2:request>
```

Listing 3.5: Fuzzino SQL request example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:response xmlns:ns2="http://fuzzino.fuzzing.fokus.fraunhofer.de/response">
    <string id="220d0a9c-7d63-4e50-a17a-245567ddaa72" moreValues="true" name="SQL">
        <generatorBased>
            <generator name="SQLInjections">
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">1;SELECT%20*</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">';DROP users;--</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">' or 1=1--</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">') or ('1'='1--</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">';shutdown -- </value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">;SELECT LOAD_FILE(0x633A5C626F6F742E696E69)
                        ;--</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                ...
            </generator>
            <generator name="SQLTimeInjections">
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">1 or sleep(1)#</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">;waitfor delay '0:0:1'--</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">1 or benchmark(10000000,MD5(1))#</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                <fuzzedValue>
                    <value xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org
                        /2001/XMLSchema" xsi:type="xs:string">'; if not(substring((select @@version),25,1)
                        &lt;&gt; 5) waitfor delay '0:0:1' --</value>
                    <kind>GENERATED</kind>
                </fuzzedValue>
                ...
            </generator>
        </generatorBased>
    </string>
</ns2:response>
```

Listing 3.6: Fuzzino SQL response example

Radamsa

Radamsa [9] is a basic black-box mutational fuzzer. It's a command-line tool that takes input and then uses random mutation. It has no knowledge of the input structure and input boundaries, but it can derive some structural properties from the supplied input. See Listing 3.7 for and example. From the given input "HAL 9000", it can derive that "HAL" is a string and "9000" is an integer, separated by a space. When mutating, it sometimes changes the integer value, sometimes the string value, and sometimes both. To allow further control over the mutation process, various parameters can be supplied. By default, it chooses one or more random mutation strategies, but this can be influenced, and the number of mutations per input can be set. Furthermore, the 'seed' parameter can be used to make the output deterministic, for reproduction purposes.

```
1  echo "HAL 9000" | radamsa
2  HA$(xcalc)%d&#0000;+inf;xcalc%p;xcalc\u0000'xcalc%#x$‘;xcalc%d+inf'xcalc\u0000&#000;\x0a$'
3
4  echo "HAL 9000" | radamsa
5  HAL 99223372036854775807
6
7  echo "HAL 9000" | radamsa
8  HAL -1
9
10 echo "HAL 9000" | radamsa
11 HALAL\u0053 /09
12 09
13 0+0v9
14 0+0L/
15 0+0v9
```

Listing 3.7: Radamsa Output Example

Skyfire

Similar to Driller, Skyfire [29] 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 as semantic checks. Skyfire uses random generation algorithms, but uses multiple heuristics. It favors, low-probability, low-frequency, and low-complexity grammar rules, and restricts the number of rules used.

[9] https://gitlab.com/akihe/radamsa

Name	Input	View	Seed	Algorithm	Strategy	Deterministic
AFL	Dumb	Grey-box	Mutational	Genetic	Width	no
AFL++	Dumb	Grey-box	Mutational	Genetic	Width	no
AFLFast	Dumb	Grey-box	Mutational	Genetic	Power schedules	no
AFLGo	Dumb	Grey-box	Mutational	Genetic	Width	no
AFLSmart	Smart	Grey-box	Generational	Genetic	Width	no
Angora	Dumb	Grey-box	Mutational	Genetic	Width	no
Driller	Dumb	Grey-box	Mutational	Concolic Execution	Undiscovered paths	no
FairFuzz	Dumb	Grey-box	Mutational	Genetic	Rare branches	no
Fuzzino	Smart	Black-box	Both	Multiple	Multiple	yes
honggfuzz	Dumb	Grey-box	Mutational	Genetic	Width	no
libFuzzer	Dumb	Grey-box	Mutational	Genetic	Width	no
Nautilus	Smart	Grey-box	Generational	Multiple	Depth	no
Peach	Smart	Black-box	Both	Multiple	Multiple	yes
Radamsa	Dumb	Black-box	Mutational	Random	Multiple	no
SlowFuzz	Dumb	Grey-box	Mutational	Genetic	Performance-based	no
Skyfire	Smart	Black-box	Generational	Random	Low-frequency	no
Superion	Smart	Grey-box	Mutational	Genetic	Width	no

Table 3.1: Comparison Matrix of Selected Fuzzers

Chapter 4

MOfuzz and Fuzzion: the offline versus online frameworks

In this chapter, we explain the two different architectures for our implementation: the offline approach and the online approach. Then we explain how we implemented this architecture using multiple fuzzers, what design decisions we made and what the limitations are. In Section 4.1 we discuss our architecture and implementation for the offline approach and in Section 4.2 we do the same for our online approach.

As stated in Section 3.1, there are two possible approaches to combine fuzzing with MBT. An offline approach takes an existing fuzzer, and supplies it with model information, while an online approach uses fuzz data generators and integrates these in an existing toolset. We present two frameworks, to reflect both approaches.

4.1 Offline Approach: MOfuzz

We named our offline framework **MOfuzz**, which stands for 'Model-based Offline fuzz'. Figure 4.1 shows a full overview of the MOfuzz architecture. The complete process is supervised by a chosen fuzzer. Model information is extracted and converted into a specification that can be read by the chosen fuzzer. For instance, for Peach, this process would yield a 'pit' file, containing all label names. Similarly, for AFL, this would result in a dictionary including all label names. When a mutational fuzzer is chosen, it would require a set of sample inputs that can be mutated. For this process, we can use a test case generator in our MBT toolset to generate valid inputs and extract these to sample files.

The next step in this approach is to establish a connection between a chosen fuzzer, and the SUT. To achieve this, we need two components: a **Fuzzer Wrapper** and an **Adapter**, as shown in Figure 4.1. The Adapter is the component that is responsible for communicating with the SUT and is already present in our MBT toolset. This component has a generic interface. The Fuzzer Wrapper is a separate class or file that is specific to the chosen fuzzer. In this class, one should add all the necessary dependencies, instrumentation, and other code to make the fuzzer work with an adapter. Once this Fuzzer Wrapper has been developed, it can theoretically be used on any adapter, since the interface is generic and doesn't need modification.

Alternatively, when implementing a fuzzer that relies heavily on coverage feedback, one could choose to remove the Fuzzer Wrapper and Adapter from this setup and fuzz the SUT directly. The advantage of this approach is that the performance will be increased, and coverage feedback is enabled. However, this requires manual labor for each new SUT that is added, since it requires a specific setup to allow fuzzing. Furthermore, there could be incompatibilities between a given SUT and a fuzzer. For instance, when the SUT uses telnet or HTTP based communication, and the chosen fuzzer only supports fuzzing files or binaries. This approach lacks the feedback of model coverage, since it does not use MBT functionality to detect new paths.

4.1.1 AFL Wrapper and Dictionary Generator

We developed a generic AFL wrapper that can run on the command line, and communicates with the SUT's adapter, present in AMP. Fuzzed output values are sent to the adapter, and errors are recorded. The AMP adapter interface is generic, therefore, theoretically, any SUT modeled in AMP can be fuzzed

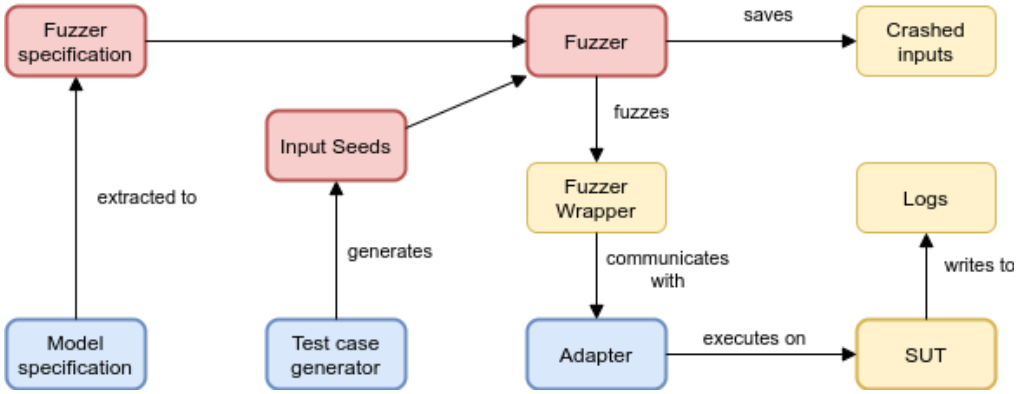

Figure 4.1: Visualization of the Offline Fuzzing Architecture, MOfuzz

using this approach.

Next, we developed an AFL dictionary generator, that extracts transition names from a given model and puts them into an AFL dictionary. AMP is written in Ruby, which is not directly supported in AFL. Therefore, to make AFL work, the Kisaten [1] package is used. This package enables instrumentation, coverage guidance and converts Ruby exceptions into AFL detectable crashes.

The dictionary generator extracts all label values for the complete model. In this approach, there is no way for AFL to know what state the SUT is currently in. Therefore, AFL could generate a valid model transition with valid parameters, but if the current state does not allow this transition, it will still get rejected.

4.2 Online Approach: Fuzzion

We named our online framework **Fuzzion**, which is a combination of 'fuzz' and 'fusion', the latter because our implementation uses a fusion of multiple fuzzers. Figure 4.2 shows a full overview of the Fuzzion architecture.

In contrast to MOfuzz, Fuzzion utilizes the complete testing process of MBT, and allows integration of fuzzers to function only as a test data generator. Fuzzion introduces a new test strategy that overlooks the fuzzing process. When Fuzzion runs a test case and encounters a label that can be fuzzed, it executes an integrated fuzzer at run-time. When executing the fuzz data generator, it is supplied with a specification and possibly a valid sample value, generated by a test case generator. This approach guarantees full model coverage, since we can reach any state by using the regular MBT functionality, and start fuzzing at any given time.

Since fuzzers typically generate mostly incorrect data, a typical test run would get stuck on an early transition in the model, because its constraints will not be met most of the time. To overcome this, we initially use a different data generator, which can generate valid inputs to traverse the model.

4.2.1 Configuration Parameters

Fuzzion contains a list of configuration parameters that should be implemented. These parameters should be configured per SUT and influence the way the SUT is fuzzed. They are used to select a suitable fuzzer, and The parameters can be altered to trigger different strategies.

- **Max number of test cases (integer)**: 0 means indefinite, to allow for continuous fuzzing and stopping on demand.
- **Max number of steps (integer)**: The maximum amount of steps per test run, these steps are performed by the MBT toolset, to traverse the model and reach states where the fuzzer can be initiated.
- **Mutate (boolean)**: This determines whether or not the label parameters should be mutated, based on a given sample value.
- **Generate (boolean)**: This determines whether or not the label parameters should be generated, based on their specification.

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

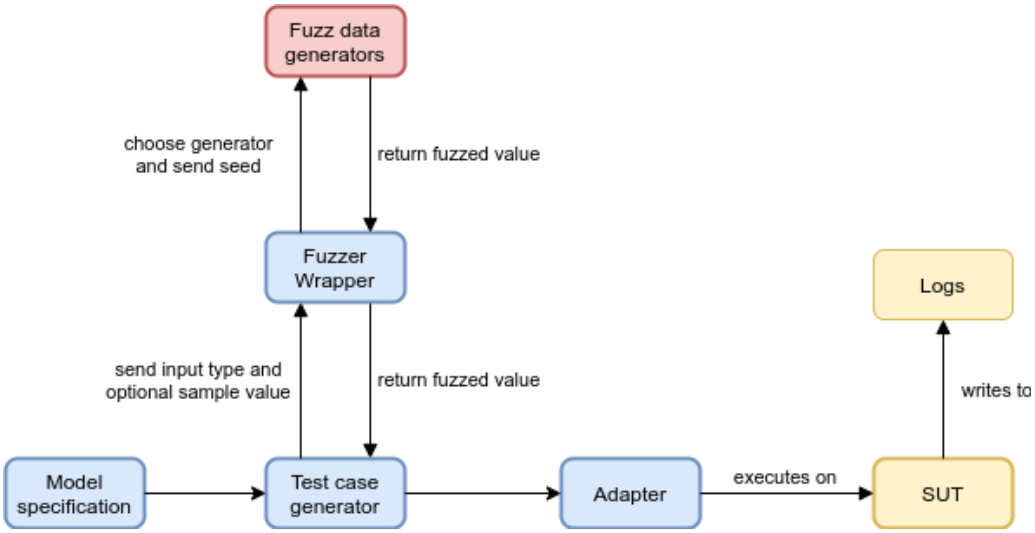

Figure 4.2: Visualization of the Online Fuzzing Architecture, Fuzzion

- **Deterministic (boolean)**: This determines the order of fuzzed inputs, when set to true, a set order will be used, when set to false, random values will be used.
- **Max fuzz tries (integer)**: The maximum amount of fuzz tries per label, a stop condition.
- **Exhaustive (boolean)**: When set to true, the fuzzer tries every generated input for each label. Only possible when using solely generated inputs.
- **Timeout (integer)**: The timeout for the system to respond when executing, in seconds.
- **Min initial steps (integer)**: Minimum amount of steps taken by the initial data generator, before fuzzing.
- **Max initial steps (integer)**: Maximum amount of steps taken by the initial data generator, before fuzzing.

At least one of **Mutate** or **Generate** should to be set to true. These parameters are used to determine which of the implemented fuzzers can be used. **Deterministic** defines the order of inputs generated for labels. When set to false, random values will be taken and used. In this case, a *Set* containing the labels and tried values will be created, and is then used to make sure the same input is not tried multiple times on a label. For this feature, we propose the *HashSet* datatype, for its fast insertion and lookup performance of $O(1)$. When the **Deterministic** parameter is configured to *true*, a fixed order of inputs will be used. When using generational fuzzers, this oftentimes means that for each mutation strategy, the size and/or invalidity of inputs is increased incrementally. For instance, when applying a "large string" strategy, the inputs will increase in size. This can be useful to systematically determine the boundaries of the system.

The **Max fuzz tries** parameter can be set to introduce a stop condition when fuzzing. This can be useful when using a (typically random) fuzzer that can generate a virtually infinite amount of inputs. This parameter is then used to make sure that when the max fuzz tries is reached for a certain label, this label will not be fuzzed anymore, and instead skips it.

For random fuzzing, the **Min initial steps** and **Max initial steps** are used to reach a random state in the model, using the initial test strategy, and then start fuzzing. This is used to achieve true randomness and to make sure the fuzzer would not always initiate on the first possible label. These parameters are ignored for exhaustive fuzzing.

For systematic generational fuzzers, the **Exhaustive** parameter can be set to true. The result of this is that the inputs generated by a chosen fuzzer will be exhausted, meaning that every input will be tried for every label. This parameter conflicts with **Max fuzz tries**, therefore, when set to true, the **Max fuzz tries** value will be ignored. Note that this will only work on fuzzers that don't generate an indefinite amount of inputs.

4.2.2 Fuzzer Interface

Our fuzzing approach is designed to support multiple fuzzers, and allow new fuzzers to be implemented easily. We designed an interface that every fuzzer implementation should adhere to. An overarching class should be developed for choosing a fuzzer, based on the set of configuration parameters. The **can_mutate** and **can_generate** methods are used to determine which fuzzers can be used, and the **has_more_values?** method can be used to determine whether a label is exhausted or not.

See Listing 4.1 for a definition of this interface in Ruby.

```
1    # @param [boolean] deterministic
2    def initialize(deterministic)
3
4    # @param [Label] label
5    def run(label)
6
7    # @param [integer] index
8    def get_value_at(index)
9
10   # @return [String]
11   def get_random_value
12
13   # @param [integer] index
14   # @return [boolean]
15   def has_more_values?(index)
16
17   # @return [integer]
18   def get_value_count
19
20   # @return [boolean]
21   def can_mutate
22
23   # @return [boolean]
24   def can_generate
```

Listing 4.1: Model-Based Fuzzer Interface in Ruby

4.2.3 Fuzz Strategy

For our Fuzzion implementation, we developed a new testing strategy in AMP, that implements our design. In all strategies in AMP, one can specify the number of test runs and the maximum amount of steps to take in a test run. Test strategies in AMP are responsible for generating input values, which are then executed on the SUT. After execution, the model will *advance* to the next state. Then the new state is compared with the state of the SUT, if this matches, the next label is chosen and processed in the same way. The Fuzz strategy, however, takes a different approach. When it reaches a label where it decides to fuzz, the inputs will be executed on the SUT, but the label will not be advanced in the model. We implemented it this way, to allow for continuous fuzzing of a single input, without having to advance to the next label after execution. If we would follow the regular flow of strategies, we would have to advance to the next label after executing every single input. This process would be inefficient and prevents us from continuously fuzzing the same label.

The downside of this approach, however, is that we can not validate that the SUT behaves in the way it is modeled. We can not perform *behavioral fuzzing*. After supplying it with inputs and without advancing, it will most certainly be in a state that mismatches the model. After initiating the fuzzer, the current test case will almost always fail. Therefore, we can only test the SUT for crashes and timeouts and not for unexpected behavior after supplying it with an invalid input.

The **Max number of test cases** and **Max number of steps** are standard parameters in AMP already, these determine the amount and length of tests cases and are typically configured based on the SUT and test strategy. We modified this to allow the value to be '0' and thus, allow an indefinite amount of runs.

4.2.4 Fuzzing Process

Using the **Min initial steps** and the **Max initial steps**, a random number 'steps_before_fuzzing' is generated for each test run. After this amount of steps, the fuzzing strategy will be activated. The *Fuzz strategy* extends from a strategy that maximizes label coverage, already present in AMP. When a test run starts, AMP will run like normal, using the default strategy. Once the 'steps_before_fuzzing' value is exceeded, every new label will be checked using the 'should_fuzz' method, which determines whether or not the current label should be fuzzed. This depends on the following conditions:

- There should be at least one label containing a parameter. When there are no parameters for a label, there is nothing to be fuzzed.
- The **Max fuzz tries** value for this label should not be exceeded, or when fuzzing exhaustively, there should be at least one input left to process.

If the 'should_fuzz' method returns 'true', the label will be passed to the Fuzzer Wrapper. The fuzzer wrapper then extracts the input type and possible sample value and passes this to the chosen fuzzer. The fuzzer returns the fuzzed value and this will be executed on the SUT. The current input value is saved in a global set of labels and inputs. This set is global to allow it to persist in concurrent test runs, to make sure the same values are not repeated in consecutive runs. When fuzzing a label, this set will be used to determine the next input. When fuzzing deterministically, only the number of previous inputs is needed to determine the next input value. When fuzzing randomly, the previous values will be used to prevent that the same value is used twice. When fuzzing randomly, the 'get_random_value' is called on each iteration, and its value is saved in the set. When fuzzing deterministically, the amount of previously fuzzed values is counted, incremented by one, and used as the parameter for the 'get_value_at' method.

Monitoring and Stop Conditions

After executing an input, the SUT's state will be checked, if it is still running and responded in time, the next value is processed. Test runs will stop when a fuzzed input triggers an exception or timeout. When this is the case, the error along with the used input is logged, and a new test run will be initiated. Another stop condition is when the current label has either exceeded the **Max fuzz tries** value, or all generated fuzzed values are exhausted. When this happens, the test run will continue like normal, but more often than not, at this point, the SUT is in a mismatched state with the model. This happens because the fuzzing process is disconnected from the regular testing process. Therefore, AMP does not process the advance of the executed label and expects the SUT to be in a previous state. Because of this mismatch, the test run will usually fail and continue to the next.

When fuzzing deterministically, a JSON file is saved to the file system, containing a mapping with the string representation of a label as key, and its corresponding iteration number as value. This file allows keeps track of the number of iterations performed, so that we can completely shut down a fuzz test and continue where we left off later. When a fuzzing run starts, this file is loaded and interpreted. See Listing 4.2 for an example.

```
{
  "?[pos]GET({\"get_variable_name\"=>\"CS_SIGN\"})": 4996,
  "?[pos]GET({\"get_variable_name\"=>\"CS_ACCNT\"})": 4996,
  "?[pos]GET({\"get_variable_name\"=>\"CS_EVENT\"})": 4996,
  "?[pos]GET({\"get_variable_name\"=>\"AI_STAMP\"})": 4996,
  "?[pos]TRANS({\"trans_id\"=>\"TM_STORE\"})": 4203,
  "?[pos]GET({\"get_variable_name\"=>\"CERTDATA\"})": 4803,
  "?[pos]STAMPREG({\"stamp_reg\"=>0})": 17,
  "?[pos]TRANS({\"trans_id\"=>\"TM_BANK\"})": 4508,
  "?[pos]TRANS({\"trans_id\"=>\"TM_CASH\"})": 3792,
  "?[pos]TRANS({\"trans_id\"=>\"TM_FLUSH\"})": 3784,
  "?[pos]TRANS({\"trans_id\"=>\"TM_BANK2\"})": 4516
}
```

Listing 4.2: JSON Iteration File Example

Timeouts

The **Timeout** parameter already exists on model definitions in AMP, and is already used in test runs. This value is used to determine whether the SUT responds in time. If this is not the case, an exception including the fuzzed parameter is thrown, which is then written to the logs, for later analysis.

In regular testing, system timeouts are caught when advancing on a label, but since the fuzzing process does not advance, we needed to implement this check in the fuzzing functionality as well. We wrapped the execution of a fuzz test in a thread and set the timeout to the configured parameter in the model. When the thread times out, an exception is thrown.

FuzzerWrapper

We designed an overarching class **FuzzerWrapper**, that is responsible for choosing and controlling fuzzers that implement the interface stated in Listing 4.1. To add a new fuzzer to this class, it needs to be registered in the constructor. On a fuzz test run, this class will be instantiated with the defined configuration parameters, and a fuzzer implementation will be chosen and constructed. Every fuzzer implementation needs to define the 'can_mutate' and 'can_generate' methods. in order for the **FuzzerWrapper** to choose a suitable fuzzer for the given **Mutate** and **Generate** configuration parameters.

When fuzzing deterministically, the chosen fuzzer will persist throughout the whole testing process. If we would let the **FuzzerWrapper** choose a new fuzzer on every new test run within the process, determinism can not be guaranteed, as inputs generated by another fuzzer differs. When fuzzing randomly, this would not be a problem, and a fuzzer will be chosen on every test run, but its generated inputs will be compared against previously processed inputs, as described before.

Note that this implementation is different from the **FuzzerWrapper** component in MOfuzz

4.2.5 Fuzzer Implementations

For this research, we implemented three different fuzzers that adhere to our selection criteria, Radamsa, Fuzzino, and Peach. In the following sections, we explain how we implemented these fuzzers.

Radamsa

Since Radamsa is a basic black-box fuzz data generator and does not support specifications, our implementation is quite basic. It takes a sample input value as argument and passes it to Radamsa, which then mutates it.

Radamsa has a 'mutation' parameter that can be set to 'mutate once', 'mutate multiple times', and 'make several mutations close by once', which can all be set. The default value in our implementation is 'mutate multiple times. By default, Radamsa can use all of its mutation strategies when mutating an input, when applicable on that certain input. This can also be controlled by specifying a list of mutation strategies in the '-m' parameter. Additionally, a 'number' mutation pattern can be set, which ensures that the output is a numeric value and not a string. This is enabled by default when the label parameter type is an integer. Even though Radamsa is dumb and random, it supports determinism by using the '–seed' parameter. When processing a string through Radamsa, and setting a seed value, the same output will be generated every time. However, the order of generated values is not systematic, there is no logical order of generated values and multiple strategies are mixed. Furthermore, this parameter does not ensure that all seeds are unique.

Peach

Peach supports both mutational as generational fuzzer. For this implementation, we wrote a Peach Pit generator, to use Peach in an online fuzzing approach. Pits are generated based on the input type and an optional value to mutate. After this Pit is generated, Peach runs it and writes values to separate files in a directory. The files are named after their iteration number, and the amount of possible inputs is determined by the last generated filename. When set to **mutate**, the directory name where these inputs are written to, is the given sample input value. When set to **generate**, the directory name is determined by its input type. This is used for caching generated inputs, since it would always generate the same values for a given Pit. When a label is fuzzed, it checks whether the directory already exists for this Pit, and generates inputs only when the directory is not present. This greatly improves the performance and allows us to continue fuzzing after we completely shut down the fuzzing process. When we set the **Deterministic** parameter in AMP to 'false', the same values will be generated by Peach, but they will be fetched in random order.

To let Peach write the inputs to files, we set the **Publisher** class to *FilePerIteration*, containing a path and a template value '{0}' that is replaced with the corresponding iteration number. When the Peach fuzzer is requested to load a value at a certain index, the corresponding file will be read and

returned. Peach is also capable of writing generated values to the console, which can then be read on the fly. We found that this greatly reduces performance, as all values should be read and saved in memory. If we would use the console to generate a single value on the fly, we still have significant latency, since Peach has to be run on every input. Therefore, we decided to perform a single run on every label and cache its values, since this offers the best performance. When the directory already exists when requesting an input, Peach will not run, and the value will be returned. When the directory does not exist, it is created and filled by running executing the generated Pit. Another benefit of saving inputs to files is that when the complete testing process terminates, all generated values will persist on file storage until they are explicitly deleted. See Listing 4.3 for a generated example for a sample string value.

When the **Strategy** class is set to **Sequential** Peach always generates a fixed number of inputs in the same order, therefore, it can be used for **Exhaustive** fuzzing.

```
<?xml version="1.0" encoding="UTF-8"?>
<Peach>
  <DataModel name="DataModel">
    <String value="HAL 9000"/>
  </DataModel>
  <StateModel name="State" initialState="State">
    <State name="State">
      <Action type="output">
        <DataModel ref="DataModel"/>
      </Action>
    </State>
  </StateModel>
  <Test name="Default">
    <StateModel ref="State"/>
    <Publisher class="FilePerIteration">
      <Param name="FileName" value="bin/data/{0}"/>
    </Publisher>
    <Strategy class="Sequential"/>
  </Test>
</Peach>
```

Listing 4.3: Generated Peach Pit for the value "HAL 9000"

Fuzzino

Similar to Peach, Fuzzino supports both mutational as generational fuzzer, we implemented both approaches. To integrate Fuzzino in AMP, we develop a wrapper in Ruby. This wrapper generates an XML **Request**, executes Fuzzino on it, and then reads the **Response**. Fuzzino format **Generators** and **Specifications** can be added to requests, and we implemented predefined **Request** classes for different input types: Command, Date, Hostnames, HTML input fields, IP addresses, Number, SQL, XML and XSS. See Listings 4.4 and 4.5 for examples. **Request** classes can contain optional and/or mandatory parameters, depending on the type of input. We also implemented a generic **String** request, which includes all possible strategies for the String type. This is useful when no contextual information is known about the input, and we only know it is of type String.

Due to Fuzzino's determinism, we needed to apply randomization to the data generation. It supports a '–seed' parameter, which skips a given amount of values. This parameter could theoretically enable randomization when given a random seed, but the problem is that we do not know the number of results before executing a request. Therefore, an IndexOutOfBoundsException can occur. Furthermore, this parameter does not seem to work in the latest version of Fuzzino. To overcome this issue, we perform requests without limiting the number of results. This impacts the execution speed, but since Fuzzino is deterministic, we can cache the requests and corresponding results. Fuzzino contains systematic determinism, which means that concurrent input values within a strategy get incrementally larger or more invalid. Fuzzino generates a fixed amount of values, which means that it can be used for exhaustive fuzzing. This follows the same format as the Peach implementation.

Our implementation caches generated values by creating a data directory per request type. In this directory, every iteration is saved to a separate file, with the iteration number as the name. Not only does this approach allow caching of generated values, but it also makes sure that consecutive fuzzing runs can continue where the previous left off, using the aforementioned JSON file shown in Listing 4.2.

Fuzzino contained a bug where some generators did not work, since the tool is open source, we sub-

mitted a pull request [2] to fix it, which was merged.

```
request = Request::Number(0, 42)
fuzzino = Fuzzino.new(request)
value = fuzzino.get_random_value
```

Listing 4.4: Fuzzino number request using the wrapper

```
request = Request::Sql
fuzzino = Fuzzino.new(request)
value = fuzzino.get_random_value
```

Listing 4.5: Fuzzino SQL request using the wrapper

4.3 Characteristics of the Offline and Online frameworks

As stated in Section 2.6, the main benefits of a framework are modularity, reusability, extensibility, and inversion of control.

MOfuzz does not completely satisfy all of these benefits. Modularity is only achieved in the adapters that execute commands on the SUT. Reusability is achieved to a certain degree, adapter components and sample generation can be reused and are generalized. Extensibility can be achieved by inheriting certain components of the framework. It lacks inversion of control because the complete process is performed by the fuzzer, rather than the framework.

Fuzzion achieves modularity by implementing the core flow of the fuzzing process while allowing the caller to specify configuration parameters. Reusability is achieved by defining a generic fuzzer interface, enabling any other fuzzer to be used in the framework, as long as it correctly implements our interface. All components of Fuzzion can be extended by inheriting them and implementing custom functionality. Fuzzion achieves inversion of control, since the complete fuzzing process is overlooked by the framework, and only the implemented fuzzer methods are called when the framework requires it.

[2] https://github.com/fraunhoferfokus/Fuzzino/pull/3

Chapter 5

Experimental Setup

In this chapter, we discuss our experimental setup for both our MOfuzz and Fuzzion implementations. Section 5.1 discusses commonly used benchmarking programs used in other research. In Section 5.2, we explain the experiments for the two approaches.

5.1 Benchmark Programs

Klees et al. [8] analyzed recent literature on fuzzing and evaluated which program types are commonly used for benchmarking. They found that the vast majority used real-world programs. Other benchmarks include Darpa Cyber Grand Challenge (CGC) [1], Google Fuzzer Test Suite [2] and LAVA-M [30]. The CGC suite contains hundreds of small programs with artificially seeded bugs. These programs are relatively small, with an average LOC of 1774. LAVA-M can artificially inject bugs into real-world software, however, the authors argue that only trivial bugs are inserted, and there is a substantial amount of future work to be done. Google Fuzzer Test Suite contains 26 older versions of real-world programs, with known bugs. The benchmarks can be run by supplying the fuzzer with a seed and then measuring the amount of time it takes before the bug is found. All of these programs are command-line tools, such as **openssl** and **libxml**. However, this suite consists solely of simple command-line-based binaries, and only supports the coverage guided fuzzers AFL and libFuzzer. This ultimately means that this suite is only suitable for benchmarking extensions or forks of one of these fuzzers, against their default baseline.

This suite heavily relies on coverage-guidance, which is not supported in our approach, due to the black-box nature of MBT. Therefore, we can safely assume that our approach would not outperform the baseline when fuzzing one of the programs in this suite. These programs are relatively simple and do not contain complex state definitions.

Another common strategy to perform fuzzing experiments is to test it on real-world software. Since the incompatibility between the aforementioned suites and our implementation, we decided to perform experiments on an existing real-world program that is already modeled in AMP.

5.2 Experiments

In this section, we discuss the corpus that we used for our experiments, and how we set up the experiments for our two approaches. All our experiments were performed on a Lenovo Thinkpad T470s laptop, with an Intel i5 CPU and 20 GB of RAM, running Ubuntu 18.04. All runs were performed on a single CPU thread.

5.2.1 Corpus

For our experiments, we use a simulator of the ITAB [3] Scanflow Point of Sale (POS) system.

The interface of the simulator is written in Python and uses a telnet based interface. See Figure 5.1 for a full visualization of the model. This system contains 5 *GET* labels with string parameters, 5 *TRANS* labels with string parameters and 1 integer label *STAMPREG*.

[1] https://github.com/CyberGrandChallenge
[2] https://github.com/google/fuzzer-test-suite
[3] https://itab.com/

To allow the Scanflow POS system to be fuzzed, we applied some changes to the setup. By default, the system would not start and stop automatically, this was a process done by hand, in a console instance. Using regular functional testing in AMP, it rarely occurs that the complete instance crashes. Therefore, we automated this process by implementing **start**, **stop**, **restart** and **is_running** methods, that are used in test runs. This way, we can determine whether an input crashed the SUT, and restart it when necessary.

5.2.2 MOfuzz Experiments

Klees et al. [8] proposed guidelines for fuzzing experiments using AFL. The authors argue that fuzzing runs should at least run for 24 hours. Their results show that comparisons with a smaller time-frame can yield false results since the fuzzers performance can vary greatly within this time-frame. Furthermore, the authors argue that multiple fuzzing runs using the same configurations should be performed. This is due to a combination of the randomness of AFL's seed selection, and the coverage feedback. Using random seeds, one path can be traversed in one run, where another run could take a completely different path. Our approach does not support coverage-guidance, however, we argue that this statement still holds, because the randomness of AFL persists in our approach. Therefore, we performed two 24 hour runs.

By default, AFL does not work when trying to fuzz a shell script (including Ruby) and only supports compiled programs. AFL states that fuzzing shell scripts slows down the fuzzing process by a factor of 20 or more. To overcome this, we needed to patch AFL to accept fuzzing shell scripts.

For our experiments, we used AFL version 2.52b. We used our dictionary generator to extract label names, and we created sample seeds for all labels. The first command that needs to be performed in the Scanflow POS system is the login. To prevent AFL from being stuck on the login functionality for most of the time, we log it in automatically after each start.

5.2.3 Fuzzion Experiments

All our experiments for Fuzzion used a **max_tries** of 10000. Note that this value is not used in the exhaustive experiments. For these experiments, we included the actual number of tests in the Results section. The timeout to process and input is set to 1 second, which was already set in the model definition.

Fuzzer Configurations

All implemented fuzzers are used in this experiment. Radamsa performs a **random** run and keeps fuzzing until the **max_tries** is reached on every label. For Peach, we included both an **exhaustive generational** and an **exhaustive mutational** run, as both these features are well supported. We performed an **exhaustive generational** Fuzzino run on the complete set of inputs. Fuzzino's mutation capabilities are very limited, we did not include a mutational run in our experiment.

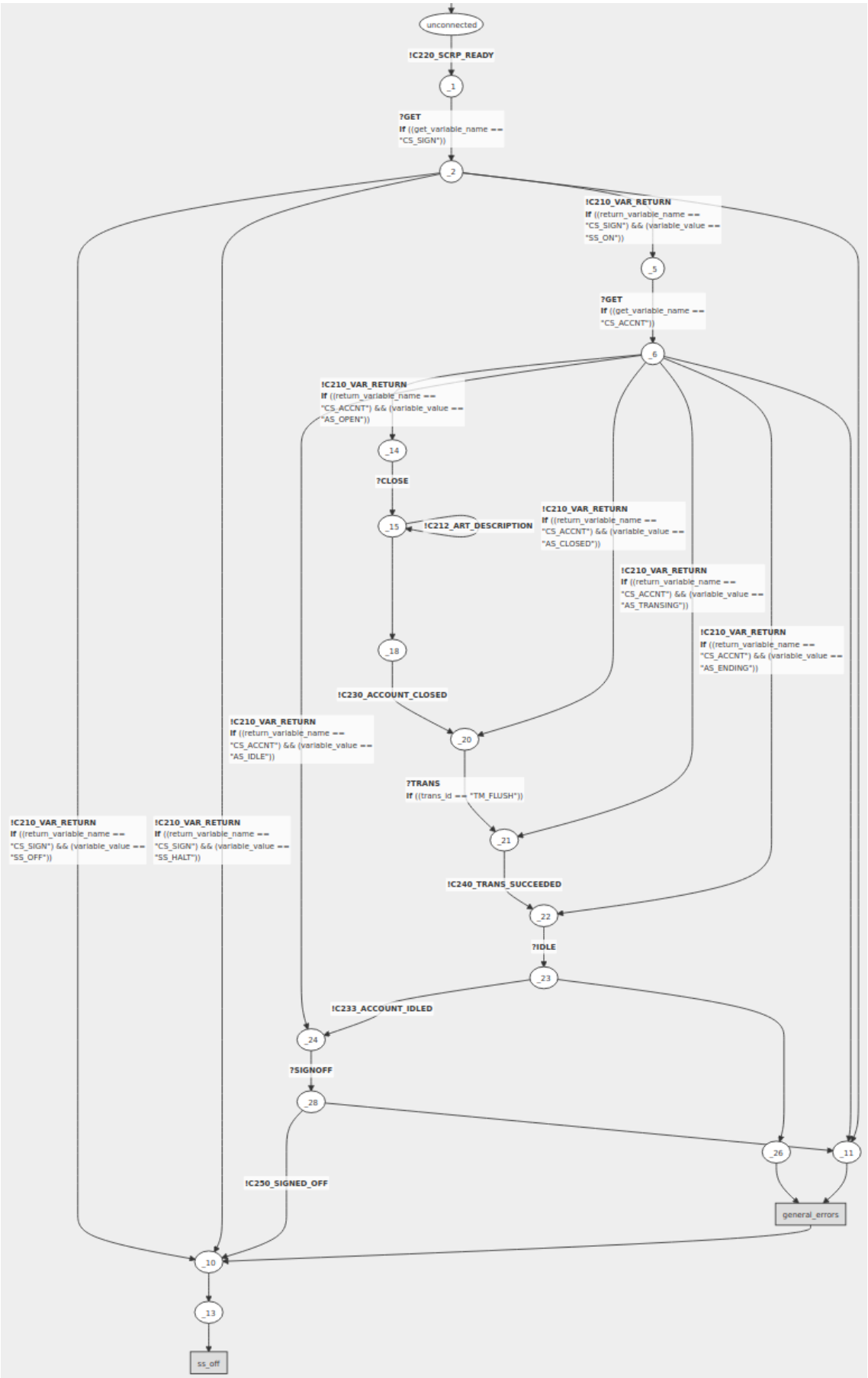

Figure 5.1: Visualization of the Scanflow POS model

Chapter 6

Results

In this chapter, we present the results of our benchmarks. In Section 6.1 we show the results of our offline approach and in Section 6.2 we show the results of our Fuzzion experiments

6.1 MOfuzz

In our first AFL run, we found a total of 1119 crashes and 125 timeouts, performing around 599000 executions. In our second AFL run, we found a total of 1104 crashes and 141 timeouts, performing around 597000 executions. The results are plotted in Figure 6.1. Both runs indicate a linear increase in the number of crashes and timeouts. We can see that over the majority of the fuzzing process, the first run did a better job in detecting crashes, but in the end, the difference is less significant.

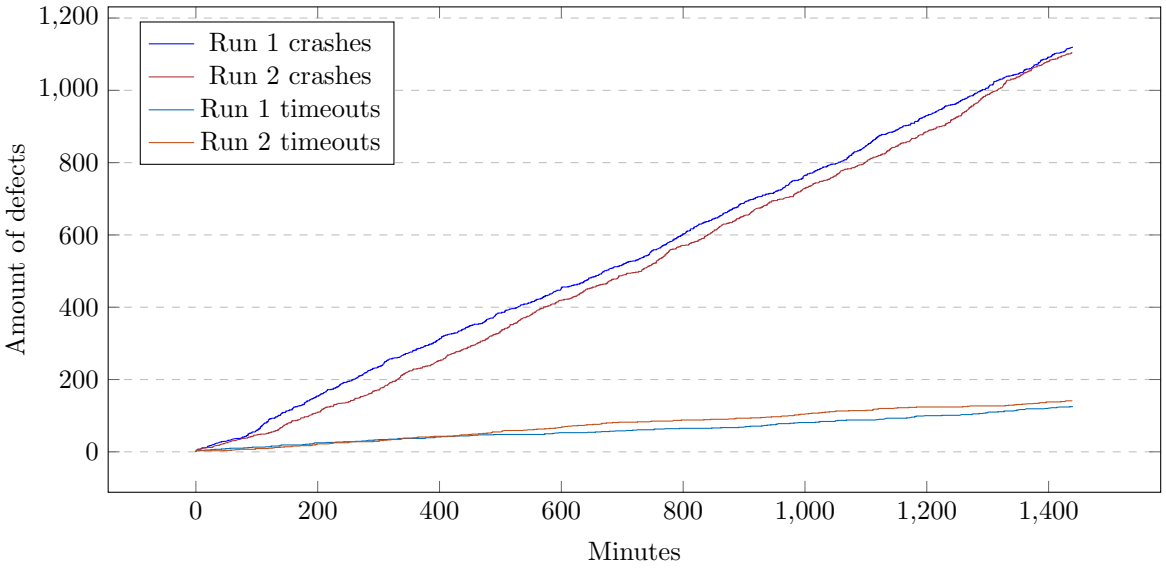

Figure 6.1: Crashes and timeouts for each 24-hour AFL run

6.2 Fuzzion

In this section, we present the results of our four Fuzzion runs. In all runs, full model coverage was achieved, and all runs resulted in varying numbers of crashes and timeouts.

6.2.1 Radamsa Random

Every label in the Radamsa Random run was fuzzed 10000 times, the maximum amount of tries we configured. The *STAMPREG* label contains an integer parameter, therefore, Radamsa is automatically instructed to treat it as a number, while mutating the value. Radamsa only generated inputs that were

successfully processed by the SUT. We did not register any crash or timeout over the population of 10000 attempts. In all the string labels, we found crashes and timeouts. This run was completed in 8 hours and 37 minutes.

During this process, we encountered numerous occasions where the testing process of AMP turned quiescent, and we had to restart the process to continue fuzzing. This occurred after registering the SUT timeout after processing exceptionally large inputs, the timeouts were logged but the process could not continue with the next input. See Figure 6.2 for a full overview of crashes and timeouts.

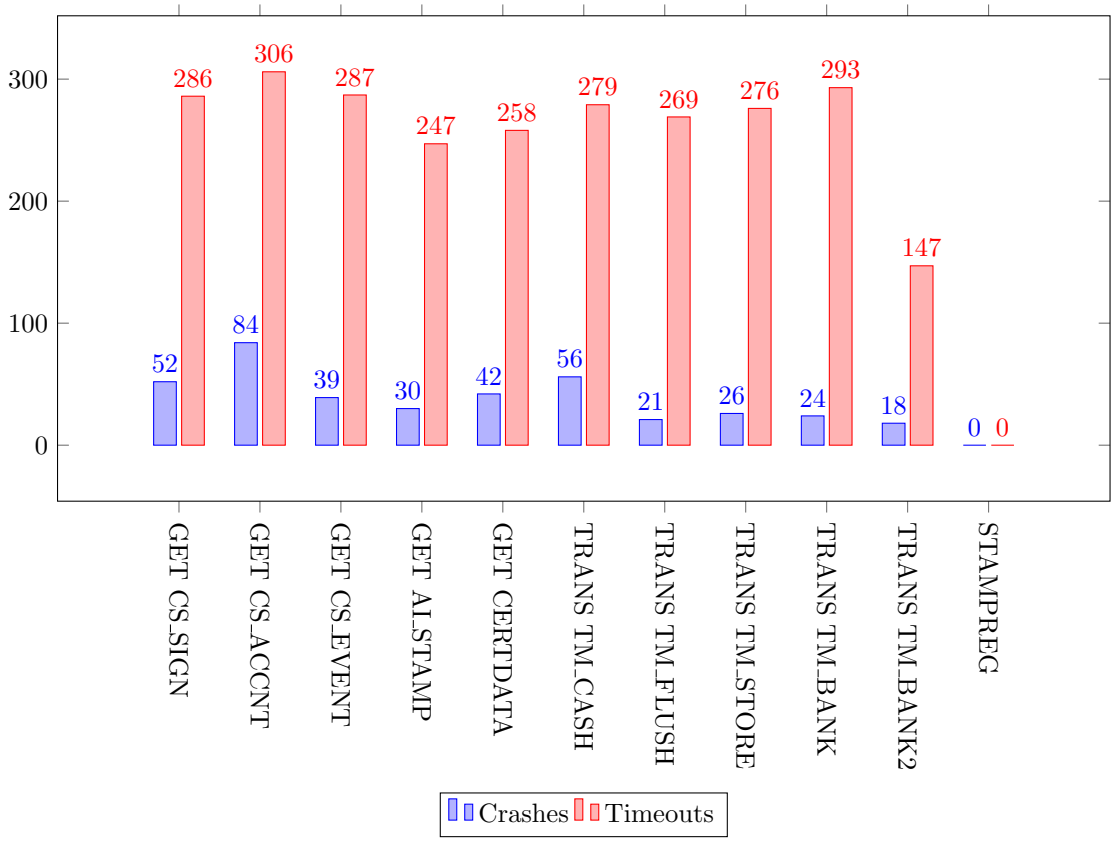

Figure 6.2: Crashes and timeouts for each label in the Radamsa Random run

6.2.2 Peach Mutational Exhaustive

In our Peach Mutational run, we set the 'exhaustive' configuration parameter to 'true'. For every label, all values generated by Peach were executed. This resulted in a total of 4298 inputs for each string label (all *GET* and *TRANS*) and 5709 for the integer label *STAMPREG*. This run took 4 hours and 46 minutes to complete. See Figure 6.3 for a full overview.

6.2.3 Peach Generational Exhaustive

Similar to our Peach Mutational run, the Peach Generational run yielded 4298 and 5709 values for the string and integer labels, respectively. We can see that the total amount of crashes over all labels are comparable with the Mutational run. However, if we compare the crashes per label with the Mutational run, we can see that this run triggers significantly more crashes on certain labels, and vice versa. This indicates that both strategies trigger different crashes per label. This run took 4 hours and 18 minutes to complete. See Figure 6.4 for a full overview.

6.2.4 Fuzzino Generational

Fuzzino generated 7135 values for each string label and 17 for the integer labels. We registered a large number of timeouts and a relatively low amount of crashes, compared to other runs. Interestingly, we did not detect any crash for all the *TRANS* labels, but between 7 and 10 for each *GET* label. Furthermore,

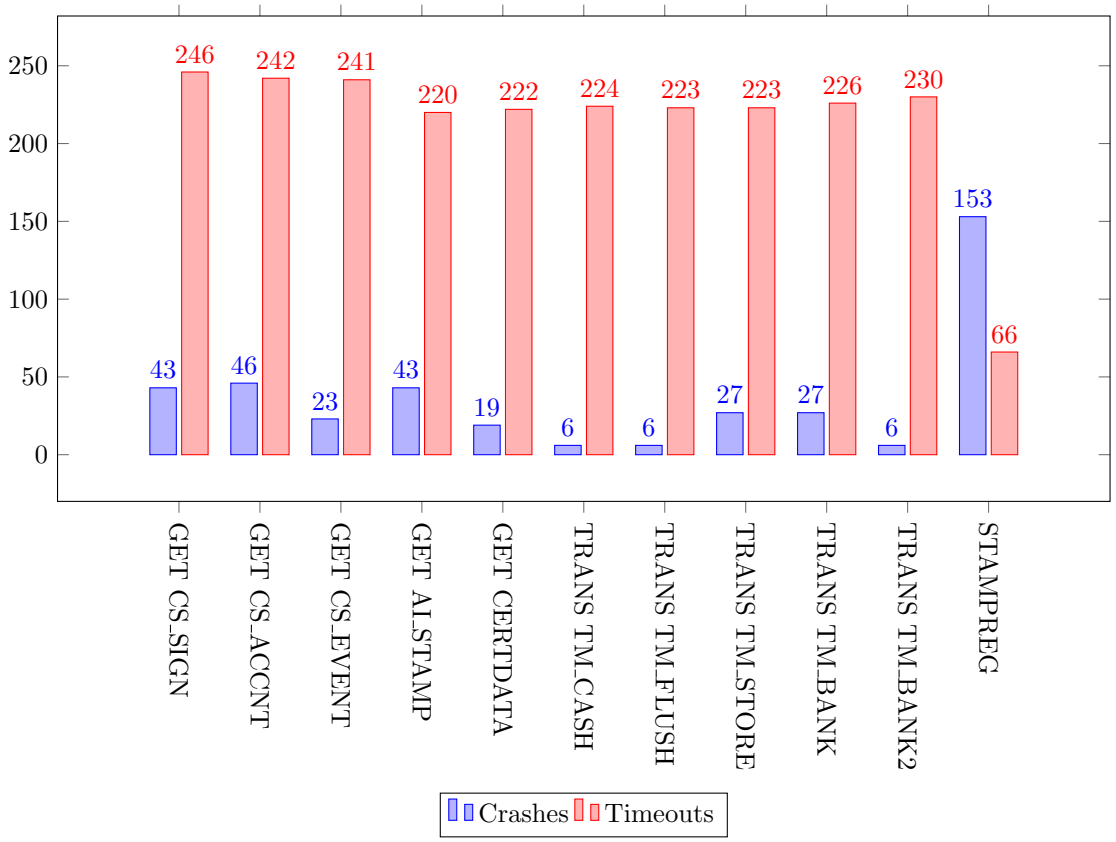

Figure 6.3: Crashes and timeouts for each label in the Peach Mutational run

no crashes or timeouts were detected on *STAMPREG*. This run took 7 hours and 5 minutes to complete. See Figure 6.5 for a full overview.

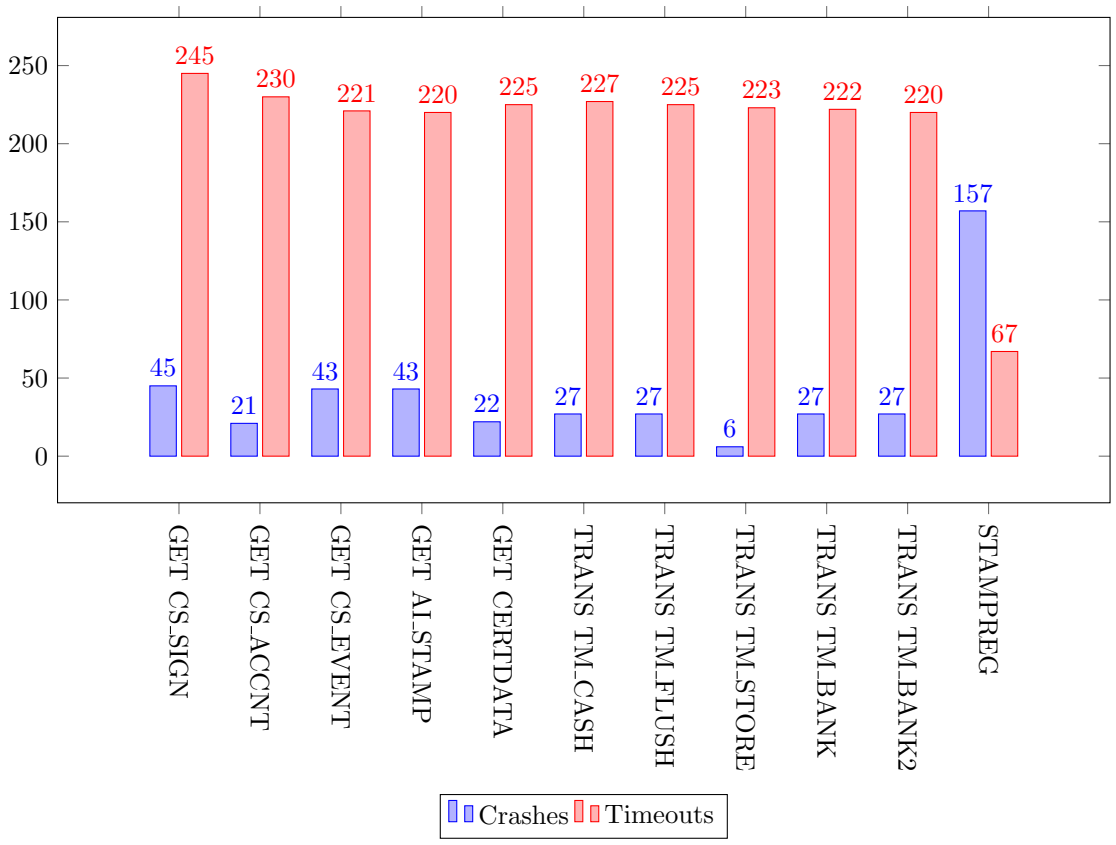

Figure 6.4: Crashes and timeouts for each label in the Peach Generational run

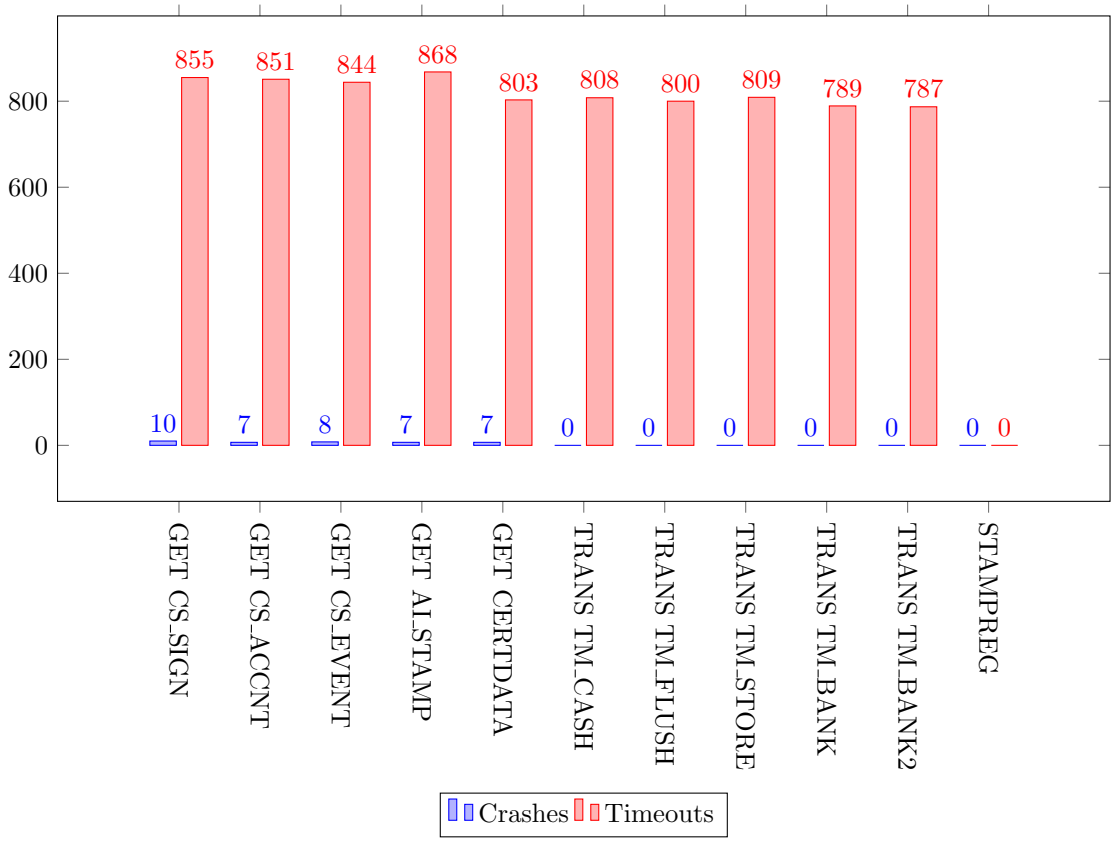

Figure 6.5: Crashes and timeouts for each label in the Fuzzino Generational run

6.2.5 Comparison

In our benchmarks, we have shown that our fuzzer implementations yield significantly different results. In both our Peach runs, a relatively large amount of crashes on the *STAMPREG* integer label were detected. Radamsa and Fuzzino did not detect any crash on this label. Fuzzino triggered a significantly higher amount of timeouts but triggered significantly fewer crashes overall. See Figure 6.6 for the full comparison.

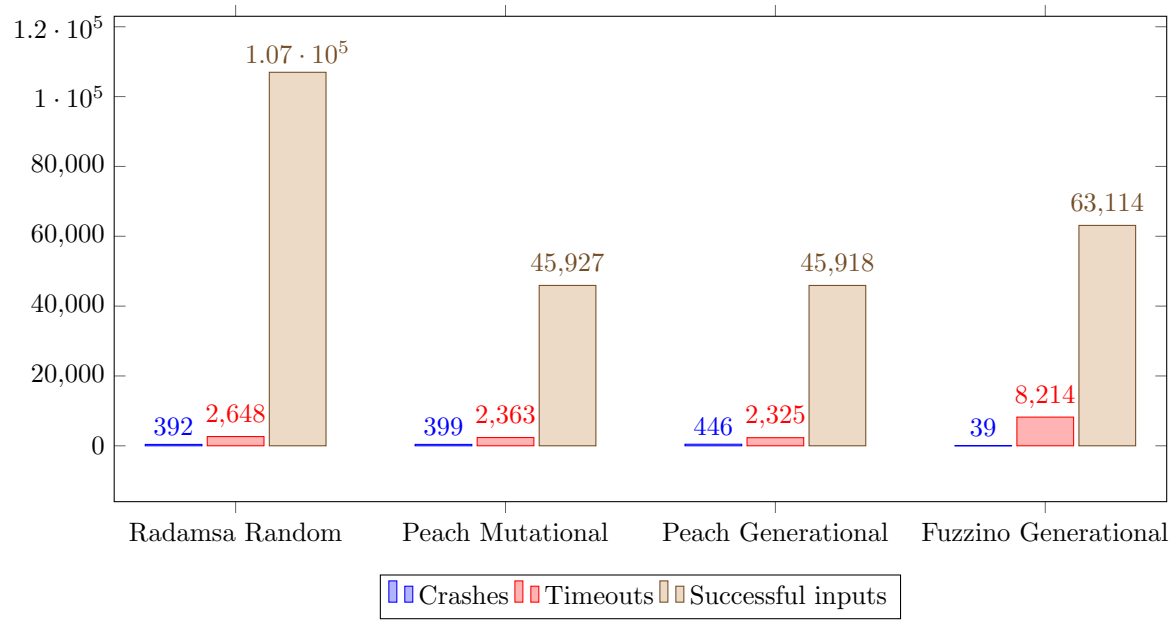

Figure 6.6: Amount of crashes, timeouts and successful inputs generated for each Fuzzion experiment performed on the Scanflow POS

Chapter 7

Discussion

In this chapter, we discuss the results of the experiments performed with the implementations of our frameworks. We discuss the implications and limitations of detected crashes and timeouts. Then we discuss the differences between random and deterministic runs. Ultimately, we answer the research questions and identify threats to validity.

7.1 MOfuzz

We have shown that MOfuzz can detect crashes. We supplied AFL with a dictionary file and valid sample seeds, which helped AFL to fuzz more effectively. This implementation lacks coverage feedback, because it only fuzzes the adapter, instead of the SUT itself. Any crash in the SUT will be linked to a single line, the line in the adapter's response method. Therefore, AFL completely lacks its coverage guidance, which essentially turns it into a black-box fuzzer.

We detected a linear increase in the number of defects found for each run. In general, the coverage-guided fuzzing runs result in a more logarithmic path, rather than linear. This could be explained by the fact that in general, unique crashes are plotted, which naturally decrease over time. Since we have no knowledge of uniqueness, and no coverage feedback, the same set of crashes can be triggered. This ultimately results in a linear increase. Coverage-guided runs would detect when a crash is a duplicate, and in that case not count it as unique. This explains the common logarithmic path. At the start, most crashes are unique, while later on, crashes will less likely be unique. Since MOfuzz does not detect this, the amount of crashes follows a more linear trend.

The combination of a lack of coverage guidance, lack of state information, mostly invalid seed label generation, and poor performance, leads to sub-optimal results for this approach. Since MOfuzz completely utilizes a stand-alone fuzzer, rather than using an MBT flow, there is no way to determine model coverage, nor do we know in which model state crashes occur.

Angora, libFuzzer, honggfuzz, and all AFL extensions work virtually in the same way as AFL, where libFuzzer and honggfuzz also support dictionaries. Therefore, we hypothesize that these tools would also suffer from the same issues as our AFL implementation.

7.2 Fuzzion

Our experiments show that the implementation of Fuzzion works as we intended in our design. All our implemented fuzzers were able to detect crashes and timeouts, with varying results. The implementation reaches full model coverage and every label can be fuzzed exhaustively. Our approach uses existing AMP functionality to transition through the SUT and can fuzz any label that contains parameters. All implemented fuzzers can theoretically be used on any SUT.

7.2.1 Crashes

We encountered a substantial number of crashes in the Scanflow POS for every fuzzer we implemented. Unfortunately, we can not show how many of these crashes are unique, since the logs indicate that almost every crash occurred on the same line in the argument parser function. Furthermore, multiple inputs could trigger the same crash, because they contain the same invalid character, while the rest of the

input string can be different. For instance, we found that a single return character '\r' turns the system quiescent, but many inputs were generated that contain this character.

For the deterministic generational runs, we notice that the number of crashes per label can vary greatly. In the Peach Generational run, we registered 6 crashes on *TRANS TM_STORE*, while we registered 45 crashes for *GET CS_SIGN*. Similarly, in the Fuzzino Generational run, we registered at least 7 crashes for all *GET* labels, while all *TRANS* labels are crash-free. In each run, the same values have been used on all *GET* and *TRANS* labels, since all of the inputs are of type String. The fact that the number of crashes for these labels varies greatly, indicates that the crashes have been triggered deeper in the system, and not at an early stage, such as argument parsing. This proves that Fuzzion can detect crashes beyond the initial stages of the program. The Peach Generational and Peach Mutational run trigger a different number of crashes per label. This indicates that there is an intersection between the strategies used for mutational and generational runs, which trigger different kinds of crashes. Therefore, it can be beneficial to perform both these types of runs, to find the most defects.

7.2.2 Timeouts

On every run, we detected a substantial number of timeouts, especially in the Fuzzino run. However, timeouts only indicate that the SUT was unable to respond within a set time frame (in this case: 1 second). A timeout in itself still does not give us much information, as it could mean one of these things:

1. The input could be processed successfully but takes longer than the set time frame.
2. The SUT needed more time to process the input, but would eventually crash.
3. The SUT turned quiescent.

In our experiments, we set the timeout to 1 second, because this was the default value for this SUT. If we want to determine the meaning of these timeouts more precisely, we could increase the timeout, and check whether the results differ. We could define an upper boundary (for instance: 10 seconds), after which we can conclude that the SUT did indeed turn quiescent. The value for the timeout would depend per SUT and we can not generalize a fixed value for this.

7.2.3 Deterministic vs. Random

Generation-based, deterministic fuzzing approaches allow for more systematic fuzzing. We can precisely determine the boundaries of input fields, since deterministic fuzzing approaches incrementally increase the size and/or invalidity of inputs. For instance, when a system can handle a string of 300 characters, but not 305, this will be easily visible by a deterministic approach, since its size is increased incrementally. When a random fuzzer determines that a system does not crash when supplying it with 42 'A's, and it crashes with 65326 'A's, more investigation is needed to determine the exact boundary. Most random fuzzers, like Radamsa, do not have test case reduction functionality. In theory, this could also solve the issue. This could be achieved by adding a test case reduction tool and trigger it at every crash. In terms of efficiency, however, random fuzzers can outperform deterministic fuzzers. They can exceed the boundary of the input more quickly, where deterministic fuzzers build the input until it becomes invalid. With the addition of a test case reduction tool, random fuzzers could theoretically outperform deterministic fuzzers. We argue that this will also depend on the SUT. When the input space is small, a deterministic fuzzer would find the boundary relatively quickly, while a test case reduction tool might need more executions to find the exact boundary. When fuzzing a system with a large input space, the performance might skew the other way.

7.2.4 Experiments

The experiment guidelines proposed by Klees et al. [8] show that comparisons that run for less than 24 hours are not thorough enough, since results can significantly change within this time frame. This mostly occurs when the coverage-guidance suddenly finds a new path and starts fuzzing previously undiscovered functionality. Furthermore, because of the random nature of fuzzing, multiple runs with the same configurations can vary greatly, therefore, multiple runs have to be performed. However, we argue that online model-based fuzzing differs greatly from this approach, since it does not rely on coverage guidance. Furthermore, this randomness does not apply when we use a deterministic fuzzer.

In our experiments, we only tested one program, which is a simulator. As stated in Section 5.2, the implementation of the SUT is hidden behind a Python interface. The error log only shows at which

line in the Python code the errors occurred. There was no way to determine in which piece of code in the implementation the error occurred. Other errors were timeouts, for which we do not have any trace of the error at all. Performing experiments using this approach on multiple programs could lead to different results. When we would fuzz a system that has a more detailed error log, we could determine the uniqueness of the crashes.

7.3 Research Questions

In this section, we answer our research questions, based on our results.

RQ1: How can we effectively and efficiently integrate fuzz testing with Symbolic Transition Systems?

Our results indicate that, on our experimental corpus, Fuzzion is more efficient than MOfuzz. The number of defects found, relative to the run-time, is higher with Fuzzion. In terms of effectiveness, Fuzzion also outperforms MOfuzz on our experimental corpus. MOfuzz detected an average of 1270 defects over an average of 598000 executions, within 24 hours. All of our Fuzzion runs detect more defects, with a smaller amount of executions, within 24 hours.

The number of crashes and timeouts found by Peach and Radamsa are comparable. However, Radamsa took almost twice as long to complete. These results show that Peach is the most effective on the experimental corpus because the total amount of defects found is higher. The efficiency is also higher because Radamsa took almost twice as long to complete. The lower efficiency of AFL and Radamsa could be due to the strategies used. Both fuzzers use relatively more small mutations (such as bitflips) than Peach and Fuzzino. This could indicate that the SUT used in our experiments is relatively resilient to such mutations.

RQ1.1: To what extent does a smart model-based fuzzer outperform a dumb fuzzer?

In our MOfuzz experiment, we implemented state-of-the-art dumb fuzzer AFL, and in the Fuzzion experiments, we used smart fuzzers Peach and Fuzzino. In terms of effectiveness, Fuzzion outperforms MOfuzz by a landslide. AFL was able to find an average of 1112 crashes and 133 timeouts in the 24 hour runs. The Peach runs yielded fewer crashes (399 and 446), more timeouts (2363 and 2325), but took significantly less time to do so. In terms of efficiency, our smart implementations outperform the dumb implementation, because more defects were found in a smaller time-frame. In terms of effectiveness, our smart implementations also outperform the dumb implementation, because ultimately more defects were found. Similar to Peach, Fuzzino also detected more defects than AFL, in a smaller time-frame. Comparing our implemented smart fuzzers to Radamsa gives a slightly different result. In terms of efficiency, the smart fuzzers outperform Radamsa. Fuzzino took less time and found more defects. The Peach runs yielded slightly fewer defects, but took almost half the time to complete. Whether all smart fuzzers outperform Radamsa in terms of effectiveness, depends on whether we relate it to the number of executions. If we only take the total amount of defects into account, Radamsa outperforms both Peach runs. However, if we relate it to the number of successful executions, Peach performs better overall.

In terms of efficiency, Fuzzion also outperforms MOfuzz in our experiment. All our online runs took 9 hours or less to complete, while we let AFL run for 24 hours. Ultimately, all smart fuzzers used in Fuzzion achieved higher effectiveness and efficiency than all dumb fuzzers.

RQ1.2: How can model information be used to define input specifications for smart fuzzers?

Models can be used to extract label information, including types, parameters, and valid sample values. This information can be used to generate a specification for smart fuzzers. This specification is then used to generate or mutate inputs accordingly. Boundary values can be extracted and passed to a fuzzer, which can then be used to effectively generate relevant invalid values. Our implementation does not yet cover more highly structured String types, such as XML or SQL. Theoretically, we can extract this information from a model, and use this to choose a suitable fuzzer that supports a given type. This allows a fuzzer to generate a specific set of values, instead of generating all kinds of values for the type String.

RQ1.3: How does the performance of generation-based fuzzing differ from mutation-based fuzzing, in terms of effectiveness and efficiency?

Our results do not indicate whether generation-based fuzzing is more effective and efficient. In the Fuzzion experiments, we performed both a mutational and a generational Peach run. While the mutational run yielded slightly more crashes, the generational run yielded slightly more timeouts. Furthermore, Fuzzino found almost 4 times more timeouts, but 10 times fewer crashes. Radamsa was outperformed by both Peach runs but performed better than Fuzzino when comparing the number of crashes.

Ultimately, our data do not indicate significantly better performance for generation-based fuzzing. So far, our results indicate that both mutation and generation-based fuzzing can be beneficial for combining with STS. The performance depends more on the fuzzer in combination with the SUT, and not whether it is mutational or generational. The performance of a fuzzer on a specific SUT depends on the strategies it uses, and how resilient the SUT is to these strategies.

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

Fuzzino and Peach allow for defining the input type, as well as its boundaries. When we fuzz integer fields, we can extract the minimum and maximum size, and use it to supply to the fuzzer. In string fields, however, we cannot simply define the type of input and its boundaries, since they do not contain information about maximum length. Furthermore, string fields do not contain contextual information, where it is used (e.g. an HTML field, XML, CLI command), nor does it contain a definition of the structure (e.g. the format used in a date field). Therefore, we can only specify the fact that the field has the string type, and let fuzzers use multiple strategies on it. When we use mutation-based fuzzers on strings, this issue could be reduced, as long as we supply it with valid sample inputs. This way, the mutation-based fuzzer can deduce some of its structural properties and use this in its mutation strategies.

RQ3: Which manual tasks can be reduced or eliminated by combining Symbolic Transition Systems with fuzzing?

We have shown that model-based fuzzing can be applied to virtually any SUT that is modeled using STS, given that it contains transitions with parameters. When implementing a fuzzer, one does not require knowledge about the SUT, since its implementation details are hidden behind adapters. Using MOfuzz, the manual task of writing a dictionary for the SUT is automated. When implementing a new fuzzer, one needs to develop an adapter for it, which can then easily be used on any SUT modeled using STS. Similarly, Fuzzion needs an initial implementation of the interface for each new fuzzer added. When a new fuzzer is added, it can theoretically be used on any modeled SUT. MOfuzz requires more manual tasks than Fuzzion. Stand-alone fuzzers typically need manual labor to implement and are not always compatible with every type of program. In our implementation, we needed to build an AFL wrapper that had to support Ruby. AFL needed heavy configuration and a separate package to work. This process took about a week to complete. In contrast, the implementation of the Fuzzion interface for Radamsa took 4 hours and for Peach and Fuzzino it took around 1 day. Furthermore, the Fuzzion interface always contains the same methods that need to be implemented, while the implementation of a new stand-alone fuzzer can require many custom configurations. Ultimately, Fuzzion reduces more manual tasks than MOfuzz.

7.4 Threats to Validity

In this section, we discuss the threats to validity we identified during this project.

7.4.1 MOfuzz Experiments

In our MOfuzz experiments, we only used the default implementation of AFL. We used AFL because it is one of the most effective and popular state-of-the-art fuzzers at the moment. However, since the performance of fuzzers can vary greatly depending on the SUT, other offline fuzzers might be able to fuzz more effectively on our corpus.

7.4.2 Uniqueness of Crashes

As stated before, our implementation lacks code coverage metrics, due to the inherent black-box nature of MBT. Ultimately, the uniqueness of crashes determines the fuzzer's effectiveness, e.g.: one fuzzer can trigger a single bug 500 times, and another fuzzer can trigger 2 unique bugs. The latter would be more effective, but in our approach, there is no generalized way to determine this. This property depends on the SUT and the way it logs crashes. If the SUT outputs stack traces, one could aggregate these, and then determine how many of these are unique.

7.4.3 Experiment Corpus

We used only a single corpus for our experiments. Due to the inherent performance variety in fuzzing, our approaches can also vary greatly, depending on the corpus used. As research has shown, different fuzzing techniques expose different types of defects, which is the reason why there exists a large number of fuzzers. A larger corpus should be used to definitively determine the optimal approach.

Chapter 8

Related work

In this chapter, we present related work on fuzzing surveys and model-based fuzzing. We group this work into two categories: surveys on fuzzing and works on model-based fuzzing. This division reflects the two main parts of our research.

8.1 Surveys on Fuzzing

In this section, we discuss related work regarding surveys on fuzzing. Table 8.1 shows a comparison of the features of these surveys.

Liang et al. [3] presented an overview of fuzzing, focused on general-purpose, as well as a classification. They classified widely used fuzzing tools, mentioned key obstacles, and included future predictions for the field. The inclusion criteria used in this research include all target programs and excludes fuzzers that have not been presented in a paper. The time range for this survey is 1990 to 2017. The comparison matrix includes target programs (e.g.: general-purpose, OS kernels, compilers), key techniques (dumb, smart, mutational/generational, key algorithm), platform, and availability (open-source, internal, closed-source).

Li et al. [4] conducted a survey, highlighting recent advances in fuzzing, which improves efficiency and effectiveness. This paper explains the fundamentals of fuzzing, including a brief history, and explains the advantages and disadvantages of common techniques. This survey, however, only focuses on highlights in new techniques, rather than presenting a complete overview of fuzzers. These selection criteria do not focus on completeness and therefore do not cover all possible fuzzers that could be used in our research.

Felderer et al. [5] conducted a systematic classification on MBST. Their classification includes a comprehensive analysis of 119 publications, published between 1996 and 2013. This paper includes all types of MBST approaches, and only includes scientific sources. Excluded fuzzers are black-box fuzzers that don't support any kind of input structure definition. Publications are classified on an extensive set of properties. The most important properties reflect the maturity of the approaches, the type of security properties that are being tested, and the evidence presented. Finally, the authors mention that there is a necessity for future classifications of the presented paper, which could include the types of target systems or the type of vulnerabilities that could be detected.

Manès et al. [7] conducted a survey 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 published between January 2008 and February 2019. The fuzzers are first grouped based on black-, grey- and white-box and the targets. Then a comparison matrix is presented, where a more detailed comparison between techniques and properties is shown. The first category is 'preprocess', this includes in-memory fuzzing, model construction, and program analysis. Then it is stated whether fuzzers use seed scheduling, followed by input generation. These categories include mutation/generation, model-based, constraint-based, and taint analysis. Then the input evaluation is expressed through crash triage: stack hash and/or coverage. Finally, configuration update properties are shown, which include evolutionary seed pool update, model update, and seed pool culling. Note that the term 'model-based' has a different meaning than the definition used in this document. The authors use the term to state that the input is represented as a model, and not that a fuzzer utilizes MBT principles. For example, Peach is considered a model-based fuzzer in this survey.

Paper	Time range	Selection criteria	Metrics and classification categories
[3]	1990 - 2017	General-purpose, presented in scientific papers	Targets, key techniques, platforms, availability
[4]	2014 - 2017	Recent advances that improve efficiency and effectiveness	Efficiency and effectiveness, black-, grey- and white-box, dumb/smart, coverage
[5]	1996 - 2013	Scientific papers, MBST in general, excludes black-box approaches	Model of system security, security model of environment, test selection criteria, maturity of evaluated system, evidence measures, evidence level
[7]	2008 - 2019	Presented at security conferences	Black-, grey- and white-box, target applications,

Table 8.1: Comparison of Surveys on Fuzzing

8.2 Works on Model-Based Fuzzing

Schneider et al. [17] 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. In this approach, results from previous runs are used to construct new seeds. This way, the test space is drastically reduced, by avoiding the execution of test cases that reveal the same error. The authors mention that after executing an invalid message, it can be unclear in what state the SUT is in. This reflects the same finding as we presented in our approach. 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 [11] is a generation-based fuzzing framework for telecommunication protocols. It is integrated into 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 a model is available, it can be fuzzed without much implementation effort. T-Fuzz uses an already defined model, written using the TTCN-3 testing language [1], which allows for easy setup and high scalability. The authors mention that the use of pre-existing fuzzing frameworks such as Peach and Radamsa is time-consuming because all the logic and protocol models need to be defined from scratch. The solution to this drawback can be compared to our Fuzzion approach, where it also used existing models and therefore allows for easy and thorough implementation. Similar to Fuzzion, T-Fuzz can be configured to perform a certain amount of initial steps, to reach a certain state, before fuzzing. The authors state that this has a significant impact on the performance of the fuzzer.

Bozic and Wotawa [2] present a model-based security testing approach where attack patterns are modeled, which can then be used to generate and execute test cases automatically. This approach uses UML state machines. In their case study, the authors show how SQL and XSS injections can be modeled and executed.

Pham et al. [31] introduced a model-based white-box fuzzing approach targeted towards binaries. The approach aims to allow fuzzing of binaries that take highly structured inputs, and where seed files might not be available. When no input seed is supplied, it uses static analysis to identify dangerous locations in the SUT. Critical locations in the SUT can be found automatically using IDAPro [2] to disassemble the program, after which lightweight analysis is performed. After determining the critical locations, test cases will be selected that are the closest to these locations.

Botella et al. [13] present an MBT approach for validating security components. The authors present three test selection criteria, that aim to cover security functional requirements, test generation, and monitoring.

BlendFuzz [32] is a model-based fuzz testing framework that targets programs that take highly structured inputs. Rather than supplying it with a pre-defined grammar, rules are inferred based on valid test cases. The approach is mutational, where test cases are mutated based on the inferred rules. Similar to many other model-based fuzzing approaches, the term 'model-based', does not refer to the same MBT

[1] http://www.ttcn-3.org
[2] https://www.hex-rays.com/products/ida

definition we use in this research. It merely refers to a set of input definitions and does not contain any state awareness.

Wang and Xiao [33] present a model-based behavioral fuzzing approach for network services. Their approach uses a finite state machine model where one can specify grammar rules, containing type, default value, and whether or not it is fuzzable.

Chapter 9

Conclusion

In this research, we defined criteria on how to combine existing fuzzing tools with MBT. Using these criteria, we reviewed recent state of the art fuzzers and composed a comparative evaluation. We defined two different approaches on how to combine fuzzing with MBT: online and offline. An offline approach relies on a fuzzer to perform the process while using information defined in models. An online approach, on the other hand, uses a fuzz data generator and fully integrates this into an MBT toolset. We developed generic architectural frameworks for both the offline and online approach, called MOfuzz and Fuzzion. For MOfuzz, we defined a generic FuzzerWrapper that can be used to fuzz virtually any SUT modeled in an MBT toolset. For Fuzzion, we define a simple interface that every fuzzer should implement. Virtually any fuzzer that adheres to our selection criteria, can be integrated when it implements this interface.

We developed a proof of concept for both the MOfuzz and Fuzzion. The MOfuzz prototype makes use of the popular coverage-guided grey-box fuzzer AFL. For this approach, we developed a dictionary generator that extracts label values from a model, which AFL can use. We extracted valid sample values for all possible labels, which AFL can mutate. For the online approach, we fully implemented the architectural design of Fuzzion, and integrated three fuzzers: Radamsa, Peach and Fuzzino. We performed experiments on a POS simulator, for both MOfuzz and Fuzzion. Our results prove that Fuzzion can detect defects using all three tools. Fuzzion outperforms MOfuzz in terms of effectiveness and efficiency. AFL needs a significantly higher amount of time and executions to find bugs. Furthermore, our results show that the performance of different fuzzers can vary greatly. Fuzzino detected mostly timeouts, but not many crashes. Radamsa was able to detect bugs but has lower effectiveness due to the larger amount of inputs generated. Peach was able to find the most crashes and greatly outperforms the other fuzzers when fuzzing integer values. These differences indicate that the integrating of multiple fuzzers into MBT is beneficial for finding different types of bugs.

While we were able to find a large number of crashes and timeouts, we can not state whether they are unique or not. Uniqueness is measured based on code coverage, which is not provided with black-box testing. Therefore, it is hard to state which fuzzer is the most effective.

9.1 Future work

In this section, we discuss multiple subjects for future work, where our current implementation is limited and/or could be improved.

9.1.1 Automatic Strategy Configuration

When generating inputs, our implementation only takes the input type into account. This is because we can currently not automatically deduce contextual information about the input value. Therefore, we can not choose suitable strategies. If we could determine where the input fields exist, e.g. an HTML form or a console input, we can use that information to supply to the fuzzer implementations. In the case of Fuzzino, this would mean that we can use one or more of the **Request** classes, instead of processing it as a generic string. This would greatly improve the efficiency of the fuzzers, since irrelevant inputs will not be generated. Furthermore, the total run time of exhaustive fuzzing would significantly be reduced.

9.1.2 Performance Optimization

Our implementation currently takes values from a fuzzer without knowledge of the used strategies and/or patterns. When the SUT crashes on a certain input, and the next input would be an increasingly large or invalid input, following the same pattern, we could skip all of the consecutive inputs, because we know that it will trigger the same bug. This process is described by Schneider et al. [17].

9.1.3 Determinism and Exhaustiveness

The Fuzz strategy contains the **Min initial steps** and **Max initial steps** parameters, which are used to generate a randomly assigned number of steps to take before fuzzing. Then, a parent strategy is used is to traverse the model, however, the goal of this strategy may mismatch the goal of our fuzzer. For instance, a parent strategy may try to maximize the model coverage, which can be achieved without fuzzing anything. Furthermore, every test run relies on randomization to reach a certain state in the model. Especially when fuzzing deterministically or exhaustively, this is not ideal, since this randomization impairs determinism. However, we can safely assume that all labels can be exhausted when we don't set a limit on the number of test cases. Ideally, the Fuzz strategy would be altered to reflect the fuzzing process more, and allow systematic testing of all labels in order, without the need for randomization and the **Min initial steps** and **Max initial steps** configuration parameters.

Our implementation of exhaustiveness checks whether a label has been exhausted, but lacks knowledge over the total progress. We can achieve complete exhaustiveness by setting the **Max Test Runs** to indefinite, but this requires manual monitoring. A better solution would be to keep track of the total amount of fuzzable labels and match this to the number of exhausted labels. When all labels have been exhausted, the test process should stop automatically. Our implementation only uses one fuzzer per test session and tries all possible values on all labels. However, other fuzzers might also be compatible with the configuration and could generate more seeds. When all seeds for a given label are exhausted by one fuzzer, another fuzzer can be used. This way, total exhaustiveness can be achieved by using all possible values from all usable fuzzers for each label on a configuration.

9.1.4 Behavioral Fuzzing

The Fuzz strategy is currently only concerned with triggering crashes or timeouts, however, when it doesn't trigger errors, the model state is mismatched with the state of the actual SUT. Therefore, we can not validate the behavioral resilience of the SUT, e.g.: when the SUT did not crash after processing an invalid input, it may still be in an invalid state. In our current approach, we can not validate this, because we don't advance the model when after fuzzing. Schneider et al. [17] described the approach of behavioral fuzzing using MBT.

9.1.5 Full Integration into AMP

Currently, crashes and timeouts are written to logs and reported back to AMP. However, to see which inputs crashes on which label, one has to browse through all test runs separately. It would be more convenient to have a separate page in AMP where all crashed inputs for each label are listed, along with the error message reported back by the SUT.

9.1.6 Integration of Known Vulnerability Databases

Though fuzzers usually generate test cases based on vulnerability heuristics, it could be useful to integrate databases with known vulnerabilities in a real-time fashion. Such databases include National Vulnerability Database (NVD) and the Common Vulnerabilities Exposure (CVE) database. This way, we could systematically test systems using the most recent known vulnerabilities.

Bibliography

[1] J. Tretmans, "Model based testing with labelled transition systems", in *Formal Methods and Testing*, vol. 4949, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38, ISBN: 978-3-540-78916-1 978-3-540-78917-8.

[2] J. Bozic and F. Wotawa, "Security testing based on attack patterns", in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, OH: IEEE, Mar. 2014, pp. 4–11, ISBN: 978-1-4799-5790-3.

[3] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art", *IEEE Transactions on Reliability, VOL. 67, NO. 3*, 2018.

[4] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey", presented at the Cybersecurity, Springer, 2018.

[5] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, "Model-based security testing: A taxonomy and systematic classification", *Software Testing, Verification and Reliability*, vol. 26, no. 2, pp. 119–148, Mar. 2016, ISSN: 09600833.

[6] C. Chen *et al.*, "A systematic review of fuzzing techniques", *Computers & Security*, vol. 75, pp. 118–137, Jun. 2018.

[7] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey", 2019.

[8] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing", *2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[9] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing", *Symposium on Network and Distributed System Security (NDSS)*, Mar. 2017.

[10] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for software security testing and quality assurance*, ser. Artech House information security and privacy series. Norwood, MA: Artech House, 2008, 287 pp., ISBN: 978-1-59693-214-2.

[11] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols", in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, Cleveland, OH, USA: IEEE, Mar. 2014, pp. 323–332, ISBN: 978-1-4799-2255-0.

[12] M. Felderer, B. Agreiter, P. Zech, and R. Breu, "A classification for model-based security testing", in *VALID 2011 : The Third International Conference on Advances in System Testing and Validation Lifecycle*, IARIA, 2011, p. 6, ISBN: 978-1-61208-168-7.

[13] J. Botella, J.-F. Capuron, F. Dadeau, E. Fourneret, B. Legeard, and F. Schadle, "Complementary test selection criteria for model-based testing of security components", *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 4, pp. 425–448, Aug. 2019, ISSN: 1433-2779, 1433-2787.

[14] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-based security testing", *Electronic Proceedings in Theoretical Computer Science*, vol. 80, pp. 1–12, Feb. 27, 2012, ISSN: 2075-2180.

[15] M. Utting, B. Legeard, F. Bouquet, E. Fourneret, F. Peureux, and A. Vernotte, *Recent Advances in Model-Based Testing*. Elsevier, 2016.

[16] L. Frantzen, J. Tretmans, and T. A. C. Willemse, "A symbolic framework for model-based testing", in *Formal Approaches to Software Testing and Runtime Verification*, vol. 4262, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 40–54, ISBN: 978-3-540-49699-1 978-3-540-49703-5.

[17] M. Schneider, J. Grossmann, I. Schieferdecker, and A. Pietschker, "Online model-based behavioral fuzzing", in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, Luxembourg, Luxembourg: IEEE, Mar. 2013, pp. 469–475, ISBN: 978-0-7695-4993-4 978-1-4799-1324-4.

[18] J. Zhao and L. Pang, "Automated fuzz generators for high-coverage tests based on program branch predications", in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, Guangzhou: IEEE, Jun. 2018, pp. 514–520, ISBN: 978-1-5386-4210-8.

[19] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing", in *IEEE Transactions on Software Engineering*, IEEE, 2019.

[20] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities.", presented at the CCS 2017, ACM, 2017, ISBN: 978-1-4503-4946-8.

[21] M. E. Fayad and D. C. Schmidt, "Object-oriented application frameworks", *Communications of the ACM*, pp. 1–7, Oct. 1997.

[22] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing", arXiv, Jan. 2019.

[23] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars", *Network and Distributed Systems Security (NDSS) Symposium 2019*, 2019.

[24] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain", in *Proceedings of the 2016 ACM SIGSAC Conference on computer and communications security*, ACM, 2016, pp. 1032–1043.

[25] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols", presented at the CCS'17, October 30-November 3, 2017, Dallas, TX, USA, ACM, Oct. 2017, ISBN: 978-1-4503-4946.

[26] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search", in *39th IEEE Symposium on Security and Privacy*, IEEE, 2018.

[27] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Superion: Grammar-aware greybox fuzzing", NDSS, 2016.

[28] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage", in *Proceedings of the 33st IEEE/ACM International Conference on Automated Software Engineering - ASE 2018*, ACM Press, 2018.

[29] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing", *2017 IEEE Symposium on Security and Privacy*, 2019.

[30] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition", *2016 IEEE Symposium on Security and Privacy*, 2016.

[31] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries", in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, Singapore, Singapore: ACM Press, 2016, pp. 543–553, ISBN: 978-1-4503-3845-5.

[32] D. Yang, Y. Zhang, and Q. Liu, "BlendFuzz: A model-based framework for fuzz testing programs with grammatical inputs", in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, Liverpool, United Kingdom: IEEE, Jun. 2012, pp. 1070–1076, ISBN: 978-1-4673-2172-3 978-0-7695-4745-9.

[33] J. Wang, T. Guo, P. Zhang, and Q. Xiao, "A model-based behavioral fuzzing approach for network service", in *2013 Third International Conference on Instrumentation, Measurement, Computer, Communication and Control*, Shenyang, China: IEEE, Sep. 2013, pp. 1129–1134, ISBN: 978-0-7695-5122-7.