Institute of Software Engineering
University of Stuttgart

Universitätsstraße 38
D-70569 Stuttgart

Bachelor Thesis

The Application of Fuzzing in Software Product Line Testing: Identifying Product-Specific goals for Guiding a Fuzzer based on Variability Models

Dominik Aust

Course of Study: Informatik

Examiner: Prof. Dr. Stefan Wagner

Supervisor: Dr. Halimeh Agh

Commenced: April 3, 2023

Completed: August 27, 2023

Abstract

Fuzz-Testing or Fuzzing is a testing approach based on the semi-random provision of inputs for software systems under test. With specific adaptations, the election of inputs can be manipulated in order to optimize the testing performance. The field of fuzzing features many publications which introduce distinctive fuzzing algorithms optimized for specific products or with general advancements. A Software Product Line (SPL) describes a family of software products which share core properties but consist of diverse configurations and combinations of features. In this thesis, we introduce a process for the analysis and augmentation of variability models of SPLs which allows the product-specific optimization of fuzzing programs for these particular products. We apply the proposal to an examplary SPL and test a product instantiation to evaluate the effectiveness and efficiency of the adapted fuzzer compared to domain-independent fuzzers. Due to the fluctuating results of fuzzing, caused by the semi-randomness of input election, we perform numerous tests examining the minimum-, average- and median amounts of testing rounds, required to find different bugs and the relative rate of success of finding the bugs. The evalution shows that the test results confirm the increased performance of our novel approach in testing instantiations of the particular SPL, when compared to domain-independent approaches.

Abstract in German (Kurzfassung)

Fuzz-Testing oder Fuzzing ist ein Testansatz, der auf der semi-zufälligen Bereitstellung von Eingaben für zu testende Software-Systeme basiert. Durch spezifische Anpassungen kann die Auswahl der Eingaben beeinflusst werden, um die Performanz zu optimieren. Veröffentlichungen in diesem Gebiet stellen zahlreiche Fuzzing-Algorithmen vor, die für spezifische Produkte optimiert sind oder allgemeine Verbesserungen aufweisen. Eine Software-Produktlinie (SPL) beschreibt eine Familie von Softwareprodukten, die gemeinsame Kernmerkmale teilen, aber aus verschiedenen Konfigurationen und Kombinationen von Funktionen bestehen. In dieser Arbeit stellen wir einen Prozess zur Analyse und Erweiterung von Variabilitätsmodellen von SPLs vor, welcher die produktspezifische Optimierung von Fuzzing-Programmen für enthaltene Produkte ermöglicht. Wir wenden unseren Prozess auf eine beispielhafte SPL an und testen eine Produktinstanz, um die Effektivität und Effizienz des angepassten Fuzzers im Vergleich zu domänenunabhängigen Fuzzern zu bewerten. Variable Ergebnisse des Fuzzings, die durch den Zufallsfaktor in der Eingabenauswahl entstehen, erfordern die Durchführung mehrerer Tests. Die Vergleichskriterien sind die minimale-, durchschnittliche- und der Median der Anzahl von Testrunden, die erforderlich sind, um verschiedene Fehler zu finden, sowie die relative Erfolgsrate des Auffindens von Fehlern. Die Auswertung der Testergebnisse bestätigt die erhöhte Performanz unseres Prozesses beim Testen von Instanzierungen der zugrundeliegenden SPL, verglichen mit den domänenunabhängigen Ansätzen.

Contents

1 Introduction **15**
1.1 Context and Motivation . . . 15
1.2 Objectives . . . 16
1.3 Methods . . . 16
1.4 Thesis outline . . . 17

2 Fuzzing **19**
2.1 Mutation and Generation . . . 19
2.2 Knowledge about the SUT . . . 20
2.3 Guidance . . . 22
2.4 Available Fuzzing Tools . . . 31

3 Variability Models in SPLE **33**
3.1 Feature Model . . . 33
3.2 Variability in Space and Time . . . 36
3.3 More Variability Models . . . 38
3.4 Combined Variability Model . . . 38

4 Optimization of Fuzzing for a SPL **41**
4.1 Creation of a complete Feature Model . . . 41
4.2 Augmentation of the Feature Model with Categories . . . 42
4.3 Augmentation of the Feature Model with Variability in Time . . . 43
4.4 Assignment of Priorities to the Features and their respective Components . . . 44
4.5 Methodology of our Approach . . . 47
4.6 Insertion of Instrumentation to collect Runtime Data . . . 48

5 Evaluation **51**
5.1 Realization of the Fuzzing Project in Java . . . 51
5.2 Comparison Criteria . . . 54
5.3 Fuzzing Results under a constant Input Order . . . 55
5.4 Effect of Other Factors than a and b . . . 64

6 Conclusion **69**

Bibliography **71**

A Appendix **75**
A.1 Fuzzing Results under random Input Orders . . . 75

List of Figures

2.1 An illustration of Blackbox-Fuzzing - the fuzzer has limited access and knowledge to the SUT. 21
2.2 An illustration of Whitebox-Fuzzing - the fuzzer can utilize information about internal SUT modalities. 21
2.3 JPEG images "produced from scratch"[Zal14]. 24

3.1 The operators of feature models. Summarized from [LLLS17; Mar06]. 34
3.2 The feature model of the BCS by Lity et al. [LLLS17] 35
3.3 The earlier described variability of the fictional washing machine illustrated in a variability model. The syntax is inspired by Wittler et al. [WKR22]. 37
3.4 Our simplified variability model - the base for product-specific fuzzing optimization. Inspired by the Refined Conceptual Model by Wittler et al. [WKR22]. 39

4.1 A simple yet complete feature model of the BCS. Adapted from [LLLS17]. . . . 42
4.2 Optimizied clustering and categorization in the feature model of the BCS. Adapted from [LLLS17]. 43
4.3 Determination of testing categories in order to guide the testing specifically. Adapted from [LLLS17]. 44
4.4 An invented short example of potential variability in the BCS. 44
4.5 The highlighting of updated and new features in the feature model. Adapted from [LLLS17]. 45
4.6 Required instrumention for our approach in the syntax of Padhye et al. [PLS+19] 49

5.1 First set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 55
5.2 First set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 56
5.3 First set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 57
5.4 First set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. . . 58
5.5 Second set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 59
5.6 Second set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 59
5.7 Second set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 60
5.8 Second set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. . . 60

5.9 Third set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 61
5.10 Third set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 62
5.11 Third set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 63
5.12 Third set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. . . 63
5.13 Role of c: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 64
5.14 Role of c: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 65
5.15 CategoryPrio: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 66
5.16 CategoryPrio: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. . . 66
5.17 ProbReset: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 67
5.18 ProbReset: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 68

A.1 First set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 76
A.2 First set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 76
A.3 First set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 77
A.4 First set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. . . 77
A.5 Second set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 78
A.6 Second set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 78
A.7 Second set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 79
A.8 Second set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. . . 79
A.9 Third set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 80
A.10 Third set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 80
A.11 Third set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. 81
A.12 Third set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds. . . 81

List of Tables

4.1 The products mapping of all existent features within the HI-category augmented with reasonable priorities. 46
4.2 The products mapping of all existent features within the DS-category augmented with reasonable priorities. 46
4.3 The products mapping of all existent features within the SEC-category augmented with reasonable priorities. 47

List of Algorithms

2.1 Pseudocode of a simple mutational coverage-based graybox fuzzer. Inspired by Padhye et al. [PLS+19]. 23

2.2 Extension of DOMAINGOAL with domain-specific waypoints. Inspired by Padhye et al. [PLS+19]. 26

2.3 Example algorithm with words consisting of a's and b's 28

Acronyms

BCS Body Comfort System. 17

CVM Combined Variability Model. 41

DS Door System. 42

HI Human Interface. 42

SAF Safety. 43

SEC Security System. 42

SPL Software Product Line. 3

SPLE Software Product Line Engineering. 33

SUT system under test. 15

1 Introduction

1.1 Context and Motivation

In Software development, the term Software Product Line (SPL) describes a volume of partly similar software products. The family of software products share some fixed core properties but can contain diverse features, described by the variability of a SPL. Benefits are for example lower development costs and increased productivity by pooling similar products and describing various options all at once instead of separately [Ins23].

Fuzz-Testing or Fuzzing is a software testing approach in which the program inputs for the system under test (SUT) are repeatedly provided semi-randomly. Fuzzing can be performed in different stages of testing like Unit Testing, Integration Testing, System Testing or Acceptance Testing. A semi-random input wrapper and the execution of a test program are determining the fuzzing characteristic. The ability to distinguish some kind of unwanted behavior from valid behavior in the SUT, is another general rule that a fuzzer must conform to. Therewith, it is possible to differentiate between faulty and valid inputs [Bet22].

In this work, we introduce fuzzing approaches and their advantages while looking at differences and similarities. We explore different variability models used in the context of SPL and extract relevant information to guide a fuzzer therefrom. The objective is to find and augment a suitable variability model for the extraction of such information and to select a fuzzer eligible for application on an exemplary SPL afterwards.

At first glance, fuzzer-provided inputs might be mostly identified as random data. Generally though, a completely random generation of inputs does not suffice to discover vulnerabilities in a reasonable amount of time or at all. That is why fuzzers either mutate formerly tested inputs under predefined mutation criteria or generate the inputs with some rules [Bet22; ZWG+21]. Consequently, it is possible to lead the development of testing in some direction. Therefore, the mutation and generation of new inputs not only revolves around random- but also intended invalid, nontypical and unexpected input data that might trigger false or unanticipated behavior. Characteristic examples for this type of misbehavior can be crashes, hangs, unavailability or effects on the usability of the SUT. Also, fuzzing might expose different effects from invalid input data that might even have a negative impact beyond the SUT itself [Yak23].

To identify potential faults in a SPL, it is not feasible to test all possible variants of the product. Potentially there might be an exponential amount of variations. Therefore, it is inevitable to follow a more general approach while testing a SPL. Finding product-specific focal testing points and therefrom choosing meaningful configurations for testing a SPL, will be examined in this work. We propose a process with the idea of improving the application of Fuzz-testing in the area of

testing SPLs. The analysis and evaluation of the optimized fuzzing process, compared to popular domain-independent fuzzing approaches, applied to the example SPL, allows conclusions about the effectiveness and efficiency of our proposal.

1.2 Objectives

In this section, we outline the three research questions accompanying our studies towards finding an improved fuzzing process for SPLs. They are supposed to structure the work and facilitate the comprehension of intermediate findings.

Research Question 1: What are the commonly used goals to guide a fuzzer?

This question investigates common approaches of fuzzers from different domains. Our objective is to introduce a selection of commonly used goals in order to create a fundamental understanding of guidance in fuzzing.

Research Question 2: Which variability model(s) are there and how can they be used to identify product-specific goals?

This question investigates commonly used variability models used in the context of SPLs. We further intend to derive product-specific goals from these models which is also part of this question.

Research Question 3: What fuzzing tools are there and which goal/goals are covered by them?

This question is generally supposed to look at different available fuzzing tools and their underlying goals used for guidance. We aim to mention examples and further literature elaborating this question in order to kick-start the election of a compatible fuzzer for operators of our process.

1.3 Methods

- We provide a general yet extensive introduction into Fuzz-testing and variability models to understand the context of this work and the necessary relations between all components.
- We present a process describing the necessary steps of analysis and augmentation of variability models in order to adapt fuzzers to the respective SPL of the model in theory. In order to practically perform the fuzzing of valid instantiations, a fuzzer capable for communication with the products interfaces is to be chosen and extended. For that, we later suggest a scientific paper which lists closed- and open-source fuzzers for inspiration and assistance. Depending on the structure and size of the SPL and the amount of configurations to be tested, the development from scratch of a fuzzer which corresponds to our recommendations could also pay off.
- In the results of this thesis, we discuss the benefits and the efforts of applying our process. We anticipate the effectiveness and efficiency of an accordingly optimized fuzzer, compared to simple domain-independent fuzzers.

1.4 Thesis outline

After the introduction, the second chapter of this work revolves all around Fuzzing. We introduce different variants and popular approaches. Also, we mention the differences of traditional fuzzing and the fuzzing of embedded systems because many SPLs consist of software and hardware parts, making it essential to understand potential difficulties like for example the simulation of hardware. Chapter 3 contains information about variability models for SPLs and ends with our Combined Variability Model (CVM) which we use for the main part of this thesis, the proposal of our optimization process. Chapter 4 is separated into the steps of this process and includes the application of each step onto the examplary SPL project, the Body Comfort System (BCS) [LLLS17]. Finally, Chapter 5 includes the evaluation of the produced test results with different kinds of diagrams.

2 Fuzzing

The origin and first occurrence of the term fuzzing can be backtraced to the University of Wisconsin. In one of his classes in 1988, Prof. Barton Miller tested several Unix programs by automatically applying random input data generated by a script. The goal was finding inputs, crashing the programs. This still resembles the modern understanding of a black-box fuzzer [TDM08].

Since then, different varieties and new possibilities of fuzzing were found and are being developed continuously. The number of programs which can be fuzzed and how efficiently they perform under test increases. In general, fuzzing has emerged and proven itself as a very effective testing technique for finding vulnerabilities in several software programs and real world applications [LZJ+19].

The original understanding of the term fuzzing mostly fits today's understanding as well. That means, Fuzzing is an automated software testing technique, used to find faults or not considered behavior in a system under test. The approach is based on the repeated generation or mutation of input data for the SUT. Input data might differ from system to system. The fuzzer needs to provide input that is somehow interpretable by the SUT. That means the input must be inserted in the right place, at the right port, so that the program can read it properly during the test.

Fuzzing is not limited to the generation of completely random data. Varieties include approaches of fuzzers with insight into program code that might be able to find crashing inputs which are very unlikely to be generated at random. The amount of such fuzzers is still increasing, as different researchers try to adapt and develop fuzzers that outperform other fuzzers in their domain and context. Oftentimes, fuzzing a SUT requires much time and many inputs to find unforeseen behavior like untypical code paths or system states that result in faults or crashes. Typically, the input triggering false behavior is saved and might help finding more bugs if the fuzzing strategy is not purely random generation of input data.

In this chapter, we introduce the different variants of fuzzers. We investigate RQ1 and RQ3 which are both related to fuzzing. The goal is to examine goals and strategies typically used in fuzzing and compare their usability, availability and performance. Moreover, in our context, we will review their general usability in the context of fuzzing embedded systems. This will be helpful for seeing which of the goals that we find and derive from variability models in the further work of this thesis, can be covered.

2.1 Mutation and Generation

The first distinction made in the mode of operation of fuzzers is their way of automatically and continuously providing inputs for the SUT. Inputs are either generated successively without making use of earlier inputs or with using earlier inputs as a base.

A generation-based fuzzer would generate each input from scratch. For testing very small programs, completely random input generation might find some bugs or unexpected system behavior due to simple inputs about which was not thought of during implementation. However, most oftenly, bigger systems reject input data that does not fulfill the systems specification with usage of an input parser. From there, as the idea behind software testing is not about providing input data that gets rejected by the system under test in the earliest stage, a generational fuzzer requires information on the input structure [Wil18].

Information specifying the header structure, data positioning in a data protocol or the input format in general is used to generate inputs that resemble valid inputs. A fuzzer that generates inputs without such information is most unlikely to find crashing inputs. Thinking for example of simple checksums, randomly calculated checksums can be dismissed without interpreting any of the actual data by a simple check. In this example, knowing how to calculate valid checksums during generation would be essential. A generational fuzzer requires a specification of the input. Therefore, a proper framework like a model or a formal grammar must be provided before utilizing the fuzzer. The fuzzer then generates random data in some parts but maintains the overall specified structure.

A mutation-based fuzzer modifies existing inputs every round instead of generating them. At the beginning a set of valid inputs is provided for mutation. As soon as mutated inputs produce positive feedback like an increase in code coverage, the input is added to the set of inputs which is also called seeds. Each round the fuzzer picks and mutates some input from the set [ZWG+21]. Other than a generational fuzzer, a mutational fuzzer does not require advanced specification of the input format. There are several methods of mutation: sequentially flipping bits with varying leaps, flipping bytes, incrementing bytes, inserting interesting integers like INT_MAX and pseudo-randomly mutating or deleting parts of the input are only some few possibilities of how a mutation could look like [Zal19].

Typically, with no deeper knowledge of the input structure, many mutations might be rejected by the SUT instantly. However, by tracking and saving increasingly improving inputs, the mutated inputs come closer to reaching the fuzzers domain-specific goal by receiving positive feedback during execution.

Hence, a mutational fuzzer is able to track the progress of provided inputs other than a generational fuzzer. Also, it might create inputs that a generational fuzzer could potentially not generate due to their specification. These could either be rejected early by the SUT but could also find actual faults as they do not follow the intended specification.

2.2 Knowledge about the SUT

For further understanding we will take a closer look into software testing in general. There are three general variants which underlie different approaches. In this classification, the amount of available information on the SUT from a testing perspective, separates the approaches.

Black-box testing is the term for software testing with no information on any internal modalities of the SUT. Figure 2.1 shows the general workflow of black-box fuzzing. The fuzzer can not utilize any information of the SUT at any time. It provides input data and the SUT produces its normal output. Depending on the output fidelity, the fuzzer might be able to interpret its current effectiveness while running but it has no influence on the output.

Figure 2.1: An illustration of Blackbox-Fuzzing - the fuzzer has limited access and knowledge to the SUT.

White-box testing refers to software testing with potentially unlimited access and knowledge about the SUT. Code level insights, resource specifications, software models and other information are additionally available besides interface specifications. A normal user would not have any knowledge of this data while utilizing the public interfaces of the SUT. This information can be instrumental in white-box testing, therewith affecting the quality and suitability of the testing environment. As shown in Figure 2.2, in white-box fuzzing, the fuzzer can make use of the open accessibility of SUT modalities. This information could be used for improved output interpretation, adapted input generation and so on.

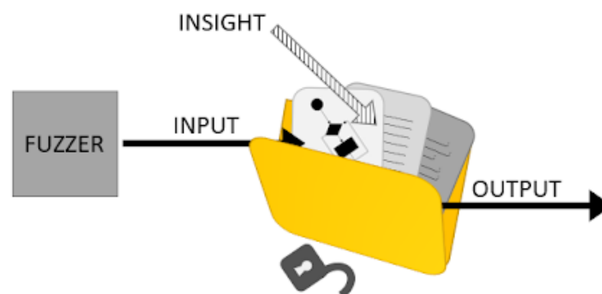

Figure 2.2: An illustration of Whitebox-Fuzzing - the fuzzer can utilize information about internal SUT modalities.

In between both extremes is grey-box testing. It can not be defined as clearly as black-box and white-box testing because it compromises the two methods loosely. As long as the information depth does not go up to the code-level, any variant with additional information can be considered a grey-box approach.

Respectively, all of these three variants can be used as a base for implementing a fuzzer. The core idea remains - that in fuzzing, there is a degree of randomness during input generation. However, there can be enormous differences in the results of a black-box fuzzer compared to a white-box fuzzer when testing the same system. That is due to the underlying functionality of input generation and mutation. Looking at the black-box approach, on the one hand, random generation without

extra interference exposes faults by pure randomness. On the other hand, deep program paths are increasingly difficult to reach without any feedback on older inputs. Especially in these program parts, a white-box fuzzer could perform better in finding bugs.

2.3 Guidance

A domain-specific goal of a fuzzer can be understood in a similar way to the metric of a search algorithm. Taking A* [Wei21] as an example: $f(x)$ is a function that returns a value from an input node x. The function estimates the distance from x to the goal node x' and by only using edges from x to nodes with a better estimate than x, there is monotonous progress towards reaching x' in A* [Wei21]. Analogously, a fuzzer with a domain-specific goal receives feedback during each round of the test, rating the usefulness of the respective input provided beforehand. Either the input helps towards reaching the goal or it does not [PLS+19]. That is the same with A*: the node x might have edges towards nodes whose metric value is worse but A* does not take these edges [Wei21]. In both situations, the intention is reaching the goal by greedily and repeatedly moving to a better state to start from in the next round [PLS+19; Wei21]. Meaning for mutational fuzz-testing under guidance that solely inputs with benefit for reaching the fuzzers goal are saved to the set of seed inputs.

Without such a metric in a search algorithm, by repeatedly using random edges, the amount of time to reach the goal node increases significantly [Wei21]. That would resemble a mutational fuzzer which saves all inputs to the set of seeds which is a very inefficient approach since many mutated inputs are rejected early or do not help with finding bugs but would still be used in further rounds in this scenario. Guidance allows a more efficient, guided approach and can be adjusted to the testers needs and to the properties of the SUT as we will see in the following sections.

Certainly, domain-specific goals and the heuristic of A* are not equal in all respects. In this comparison, the main logical difference is that a fuzzer can reuse the exact former input that was already used, compared to A* which can not go back to node x after using an edge to the next node. Now introducing this section with Algorithm 2.1, will explain the benefit of this property.

Algorithm 2.1 Pseudocode of a simple mutational coverage-based graybox fuzzer. Inspired by Padhye et al. [PLS+19].

```
1: Input: S (set of seeds), P (program under test)
2: Output: Crashes (set of crashing inputs), Hangs (set of inputs resulting in a hang)
3:
4: Crashes ← ∅
5: Hangs ← ∅
6: while true do
7:     pick ← PICKSEED(S)
8:     for i ← 1 to CALCSEED(P, pick) do
9:         mutated ← MUTATESEED(pick)
10:        program_output ← run(P, mutated)
11:        if program_output.crash then
12:            Crashes ← Crashes ∪ {mutated}
13:        else if program_output.hang then
14:            Hangs ← Hangs ∪ {mutated}
15:        else if DOMAINGOAL() then
16:            S ← S ∪ {mutated}
17:        end if
18:    end for
19: end while
20:
21: function DOMAINGOAL()
22:     if new_coverage ⊇ old_coverage then
23:         return true
24:     else
25:         return false
26:     end if
27: end function
```

Algorithm 2.1 shows the implementation of a grey-box mutational coverage-based fuzzer without knowledge about the structure of input. Several open public fuzzers with these properties exist. For instance, the very popular fuzzer AFL [Zal19] fundamentally corresponds to this algorithm. Also, the pseudocode is inspired by the explanation of Padhye et al. [PLS+19]. As we already know, a mutational fuzzer possesses a set S of seed inputs in which valid inputs provided by the user are initially stored. In general, checking for some kind of faults is the goal behind fuzzing. Therefore, Padhye et al. [PLS+19] chose to distiguish crashes and hangs by outputting separate sets for each. The contents are the exact inputs which crashed (resp. hung) the system to allow traceability. Naturally, these are empty to begin with. Within the infinite loop, one seed is picked via `PICKSEED` from the set of seeds. This happens pseudo-randomly but preferableness is possible. Let us first assume, `CALCSEED` always returns 1, therefore simplifying the nested loop. The random seed is used in `MUTATESEED` to get a mutated variant of the seed. `MUTATESEED` chooses between multiple mutation methods to apply, like bit-flipping, bit-deletion or other [Zal19]. Each of these methods uses pseudo-randomness in their mutation. After running the target program with the newly mutated input, we evaluate the result and check whether it crashed or hung the SUT during execution, saving the mutated input to the respective set in either case. Guidance comes into effect within the third branch of the if-else construct: the branch is taken when the function `DOMAINGOAL` returns true. In

the example of a coverage-guided fuzzer, it would be implemented such that it returns true iff new parts of the program were executed for the first time. Then, the mutated seed is added to the set S and will be picked by PICKSEED in some round eventually. If the mutated input does not crash or hang the system and DOMAINGOAL returns false, the input is discarded. We will further examine this in the following section.

By not deleting seeds from S, the fuzzer is able to explore the SUT under numerous inputs that were mutated on the same base. Exclusively mutating inputs in just one direction impedes the ability of finding independent program faults as it decreases input variety [PLS+19]. To avoid this, CALCSEED calculates a value for each seed and specifies how often it is mutated before randomly picking the next one from the seed set. A seed that seems probable to derive new crashing inputs or advance in the domain-specific goal, would come with a high value. That way, each seed derives several mutations within a round but can also be picked in a later round again. Usually, fuzzers attempt to discover many different crashes but there are also implementations such as in libfuzzer [Lib23], where the run would stop after the first finding of a crashing input [KRC+18].

In the following sections, we are investigating RQ1. We want to introduce the commonly used goals to guide a fuzzer and with this we introduce some fuzzing tools and their methodologies. There are domain-independent approaches which work in multiple occasions but also approaches which are specialized for specific uses.

2.3.1 Coverage-based Fuzzing

Guidance via coverage feedback works perfectly fine for many different practices. For instance, the potential was illustrated by Michal Zalewski, the developer of AFL, by feeding a text file to a JPEG decompression tool [Zal14]. He used a text file containing the word “hello” which is an invalid input that gets rejected by the input parser. By chance, or rather by stepwise coverage feedback, AFL managed to alter the header into the correct JPEG header. With gradually increasing code coverage in the SUT, AFL managed to create several valid JPEG datastreams, creating correct JPEG images eventually. This took Zalewski about six hours [Zal14]. Figure 2.3 shows numerous images produced from scratch.

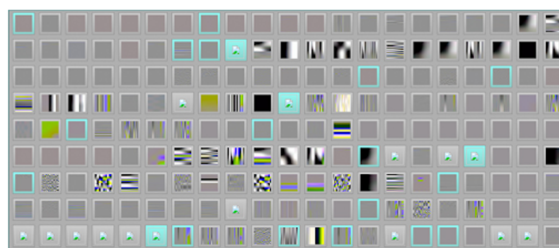

Figure 2.3: JPEG images "produced from scratch"[Zal14].

This is not a very practical example but Figure 2.3 shall illustrate how coverage-based guidance can create a multitude of inputs without knowing the grammar of a valid input. Just guided by the feedback, more and more different parts of code were addressed, showing the possibilities of mutation and the resulting broadness of seeds [Zal14].

With this variant it is possible to stress the system with unpredictable inputs that do not follow any grammar in the beginning. This might uncover unique bugs that occur if the system is run under anomalous inputs but it might also fail at an insurmountable obstacle like a difficult checksum or an encoded password [Zal14]. Also, there are more practical examples of successfully found bugs with coverage-based fuzz-testing like in SQLite [Sql23].

Technically code-coverage is the percentage of code that was executed in a test in relation to the total code of the software. The term is well-known in JUnit-testing for instance [Gri17].

An advanced method of measuring code coverage works by tracking basic block transitions. Control flow graphs make comprehension easier: the first block is a function definition. Thereafter, all sequential lines of code without decision branches and function calls belong to one block, called group statements. Comprehensively, blocks are connected to their predecessor and successor blocks or branches. A decision branch enables a block to indirectly have multiple successors: if a decision branch follows a block, the successor block is determined by the logical branch value. In a control flow graph, all successors would be connected to the branch [Bet22]. If code-coverage, hence branch-coverage increases under an input, the fuzzer would save the input and favor it compared to other seeds to more likely get picked by `PICKSEED` as in Algorithm 2.1 [PLS+19; Zal19].

In practice, the fuzzer needs to continually track these transitions in a global state. That works via instrumentation during compilation by injecting a few commands at every branch into the code. The following lines of code are injected in block beginnings in AFL [Zal19].

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

cur_location assigns a random integer to each block. shared_mem is a two-dimensional array which counts the number of every branch visited but not their global order. However, the direction in which a branch was taken can be determined due to the shift of prev_location. For instance, transitions from block A to block B and vice versa are distinct. The fuzzer maintains a global map of the shared_mem arrays from previous runs. When a mutated input holds completely new branch tuples or a higher hit count on existing tuples, `DOMAINGOAL` from Algorithm 2.1 would return true, thus saving the input [Zal19].

AFL would distinguish between the three following control flows:

1. A→B→A
2. A→B→D, because of the new tuple (BD)
3. A→B→A→B, because of the higher hit count of 2 with tuple (AB)

This advanced version of analyzing branch coverage instead of code coverage which merely checks whether a new basic block was visited, benefits the discovery of subtle faults in the SUT. For instance, security vulnerabilities are oftentimes associated with unexpected or incorrect state transitions than with simple execution of code parts [Zal19].

Coverage-based fuzzing works well in many kinds of programs due to its adaptability to the program under test. Testing increasingly new parts of a system is also a very suitable guideline for testing a system in its entirety. Also, code coverage-based fuzzers perform much better than randomized fuzzers without input knowledge and without guidance [Zal19].

In the following, we are presenting fuzzing approaches that utilize more feedback than just code coverage for choosing new seeds for their set of seeds.

2.3.2 Additional domain-specific Goals

Coverage-based fuzzing is a very popular and broad technique but for particular domains there can be different testing goals which are more specialized. With proper adaptation to the domain, the fuzzer can produce more satisfying outcomes and crash findings within the domain compared to non specialized fuzzers [PLS+19]. Recent publications from researchers and testers revolve around this topic and introduce fuzzing goals for individual domains. Exemplary goals are finding performance bottlenecks or successfully handling byte-comparisons [PLS+19]. Following such goals requires adapting the fuzzing algorithm to receive special feedback during runtime.

Thus, with appropriate monitoring and feedback control, developers are able to integrate their own goals into a fuzzer. The domain-specific orientation can pursue entirely different goals than code coverage and displace it or it might be supplementary to a coverage-based fuzzer.

In this section, we want to introduce an algorithm by Padhye et al. [PLS+19] which utilizes domain-specific waypoints in addition to the underlying coverage-based functionality. It enables to integrate existing goals from prior work as well as completely new domain-specific goals for guidance. In some contexts, the originating combinations of guidance are volitional due to the testers testing purposes or due to a potentially increased fuzzing efficiency.

We can extend our pseudocode, the Algorithm 2.1. Since this algorithm and the coverage-based algorithm in FuzzFactory [PLS+19] are mostly similar, we stick with the pseudocode and extend it with the idea of waypoints to illustrate the idea in a simple way. In FuzzFactory [PLS+19], a waypoint is an input that brings the fuzzer in a state closer to reach a specified goal and the function would need to be implemented.

Just like Padhye et al. [PLS+19], who define the function is_waypoint and use it within a decision branch after the query whether code coverage increased, we can extend the pseudocode with the functionality of is_waypoint. Hence, we would query code coverage and is_waypoint within the if-branch in `DOMAINGOAL` and could implement is_waypoint as a separate function according to the desired domain-specific behavior.

Algorithm 2.2 Extension of `DOMAINGOAL` with domain-specific waypoints. Inspired by Padhye et al. [PLS+19].

1: **function** `DOMAINGOAL()`
2:     **if** $new_coverage \supseteq old_coverage \vee$ `is_waypoint()` **then**
3:         **return true**
4:     **else**
5:         **return false**
6:     **end if**
7: **end function**
8:
9: **function** `is_waypoint()`
10:     # domain-specific implementation
11: **end function**

Padhye et al. [PLS+19]. define a domain formally as $d = (K, V, A, a_0, red)$. We will not work out the formal details of this tuple, instead we describe the preconditions and functionality. K is a set of keys that could map to program locations like basic blocks. For implementation, we would need to add instrumentation at these locations. Each location has an initial aggregate value $a_0 \in A$. The function called dsf maps a feedback value to each program location under input i. With the reducer function red, the feedback values are applied onto the current values of the locations and result in a new value from the set A for each. The reducer function must be idempotent and application-order insensitive, guaranteeing progress if an input gets to be safed [PLS+19]. is_waypoint would return true if there is progress in at least one program location k. A new input is saved when the feedback is positive in one program location.

In the following subsections, we want to give an insight into some approaches that have proven themself in the work of their respective authors, including

1. SlowFuzz by Petsios et al. [PZKJ17]
2. PerfFuzz by Lemieux et al. [LPSS18]

These distinct approaches were also introduced in Rohan Padye et al.s publication [PLS+19] with minor changes respectively as “Slow” and “Perf”. A third approach that we introduce subsequently, namely Cmp, is meant to further demonstrate the new possibilities of specific guidance in combination with a general coverage-based fuzzing approach. We are comparing these three approaches’ abilities to reach parts in the SUT beneath password checks. In this context, a password check refers to the check whether a variable and a password (constant string) are matching. Code parts after a comparison like "`if(input == password)`"for example will only be executed if the input exactly matches the password. And getting there is not trivial for a fuzzer [PLS+19].

SlowFuzz

SlowFuzz [PZKJ17] aims at maximizing the number of executed instructions or the CPU usage, in order to detect worst cases of algorithmic complexity besides finding crashes. So their effort is to find vulnerabilities in the SUT, related to an increase in runtime and resource usage. The authors introduce a very fitting example of applying it to quicksort and it describes the idea and functionality behind SlowFuzz [PZKJ17].

Quicksort has an algorithmic complexity of $O(n \log n)$ in the average case. However, with the worst possible pivot selection, the complexity is within $O(n^2)$. The worst possible pivot in every round is either the smallest or biggest element of the remaining list. Different implementations of the sorting algorithm have different methods of pivot selection but SlowFuzz would provide inputs that incrementally require more and more rounds of sorting. It will not figure out the pivot selection logically but will find worst case inputs, nonetheless.

Petsios et al. describe SlowFuzz to be a domain-independent fuzzer because users of the tool do not need any manual guidance for different domains [PZKJ17]. For us, that could be a characterization of coverage-based fuzzing, too. In our understanding, the term domain-specific refers to the ability and efficiency in finding the vulnerabilities the tester is searching for in given domains but not the necessity to manually implement the guidance in a fuzzer. For instance, domain-specific means

that in one domain the fuzzer might perform outstandingly and although it might be applicable without extra effort in another domain, it might generate insufficient test results by finding the wrong behavior or by being inefficient.

Implementing this as in FuzzFactory [PLS+19] would require a single key k. The instrumentation at each basic block simply increments $dsf(k)$ by one. With $max()$ as the reducer function, is_waypoint from Algorithm 2.2 would return true if it visited more basic blocks than all earlier inputs. Hence, it would converge to the worst case algorithmic complexity.

For finding worst cases it works domain-independently but that might not be the goal in all domains. Hence, SlowFuzz is able to find inputs that maximize execution time and it might even find an input for an involuntary infinite loop. However, it would not find a crash that occurs after a password check. It would require to mutate a seed to `password` by pure luck.

Nonetheless, when the tester wants to test the algorithmic complexity of a program and is searching for sample inputs that exploit the worst possible complexity, it is reasonable to use a fuzzer such as SlowFuzz that maximizes execution path lengths by guidance.

PerfFuzz

SlowFuzz utilizes one-dimensional feedback with its single key k. This suffices for finding long execution paths. However, it might be possible for SlowFuzz to get stuck within a local maximum of total executed blocks, thus never finding an input that reaches even worse algorithmic complexity. PerfFuzz [LPSS18] is a fuzzer by Lemieux et al. which follows a similar approach as SlowFuzz: it aims at maximizing execution counts. The difference lies within the multi-dimensionality of PerfFuzz which enables maximizing execution counts independently for all program locations [LPSS18]. Two major benefits evolve in consequence, enabling PerfFuzz to find inputs that run distinct hot spots more often and to find inputs that trigger worse algorithmic complexity than SlowFuzz by escaping local maxima in the process [LPSS18].

Implementing this as in FuzzFactory [PLS+19] would require a set of keys $\{k_1, k_2, \ldots\}$. Each basic block i maps to a key k_i and the instrumentation at each basic block simply increments $dsf(k_i)$ by one [PLS+19]. With $max()$ as the reducer function, is_waypoint from Algorithm 2.2 would return true if it visited some basic block more often than before.

Algorithm 2.3 Example algorithm with words consisting of a's and b's

1: Input: $x := (a|b)^{\geq 3}$
2: **if** $\exists n < 100 : x = a^n b a^n$ **then**
3:     **for** i **in** 1 **to** $4 \times n$ **do**
4:         $A()$
5:     **end for**
6: **end if**
7: **if** $\exists n : x = b^n a b^n a$ **then**
8:     **for** i **in** 1 **to** n^2 **do**
9:         $B()$
10:     **end for**
11: **end if**

Algorithm 2.3 explains both advantages of PerfFuzz with a simple theoretical example. The program takes a string consisting of a's and b's with a minimal length of three as input. The fuzzer is more likely to provide aba before providing $baba$ as an input since the word is shorter. Therefore, it executes block A four times and returns. After that, a fuzzer with one-dimensional feedback would not save the input $baba$ due to the lower block execution count (executes block B once). Finding the inputs $aabaa$, $aaabaaa$, ... correlates with one-dimensional guidance because more and more blocks are executed. Finding an input of the form $b^n ab^n a$ by mutation that increases the total block execution count is very unlikely. The amount of b's would need to be more than four times the amount of a's of earlier inputs to increase the block amount. Firstly, this approach won't find inputs that execute block B, least of all inputs that maximize the execution amount of block B. Secondly, the program won't discover the worse algorithmic complexity that lies within the quadratic loop. It would get caught with $a^9 ba^9$ and a block count of 36. PerfFuzz is able to develop both parts of the program step by step and individually. Seed inputs of the form $a^n ba^n$ might be mutated to $a^{n+1} ba^{n+1}$ eventually and inputs of the form $b^n ab^n a$ might be mutated to $b^{n+1} ab^{n+1} a$ eventually. $b^8 ab^8 a$ is shorter than $a^9 ba^9$ and would already have a block execution count of 64.

For finding worst cases, PerfFuzz might even outperform SlowFuzz and is able to find inputs that maximize execution times of distinct program locations [LPSS18]. However, due to the same reasons as in SlowFuzz it would still not find potential crashes after a password check. We now introduce Cmp for this purpose.

Cmp

It is much more difficult for mutational and generational fuzzers to satisfy comparisons such as '==' or '! =' than '<', '>'. Because there is only one solution to such comparisons, the fuzzer needs to guess the answer from scratch. In this approach, Padhye et al. [PLS+19] introduce a domain to make it possible for fuzzers to incrementally converge to the correct input for such comparisons with waypoints.

The idea is to save inputs to the set of seeds when they come closer to satisfying the comparison than before. That works via checking the number of common bits [PLS+19]. For instance, let zero in the binary system ('000') be the password and seven ('111') be the single initial input in the set of seeds. A five ('101') is more similar to '000' than a '111' is to '000'. So after mutating the '111' into '101', even though the password check still fails, it requires less change to get the number zero from the number five than from the number seven, making it reasonable to save '101' to the set of seeds. The number one ('001') is even closer to '000' as only one digit differs between both. If the fuzzer was only able to mutate one digit at once it would never pass the password check with just '111' in the set of seeds. This shall illustrate the idea of Cmp: saving intermediate inputs with many matching bits already, gets you closer to advancing to the exact match.

Implementing this as in FuzzFactory [PLS+19] would require a set of keys $\{k_1, k_2, \ldots\}$ with one key for every occurrence of such comparisons, functions like 'strncmp' and for switch cases. Each occurrence maps to a key k_i and the instrumentation simply stores the amount of common bits of both operands in $dsf(k_i)$ [PLS+19]. With $max()$ as the reducer function, is_waypoint from Algorithm 2.2 would return true if the amount of common bits increased at one occurrence of such a comparison.

So eventually, this approach would be able to find the crash after a password check like "`if(input == password)`". If the tester has received information that there might be password checks in the SUT this approach would be a very good choice. This is only one example of extending the coverage based approach. When encountering programs that get stuck with very low code coverage percentage it is possible to define specific waypoints in order to save intermediate inputs. While the idea originates in Padhye et al. [PLS+19]'s work, it can be generalized: it is possible to implement specific approaches for specific programs which extend the idea of code coverage. Reasonable approaches might help getting unstuck in a program location or might even increase overall efficiency [PLS+19].

In the follwing, we want to shortly address ways of guidance which are not reliant on coverage information.

2.3.3 Fuzzing without Coverage Information

The last section contains approaches which use coverage information as their first or second source of information during runtime. We want to shortly mention at this point that the term Fuzzing is not bound to coverage information. There are fuzzing approaches for particular domains or products which are not using coverage information to track their progress. Monitoring memory activities to find memory bugs, analyzing embedded system firmwares while changing environmental causalities are such examples [YRKS22]. In this context, Yun et al. [YRKS22] mention FIE [DMRJ13] and Firmadyne [CWBE16] which are examplary tools that follow these two ideas.

Guidance via basic code-coverage delivers moderate results in many domains [ZWG+21]. The aforementioned papers explicate that code-coverage is not the only choice for all applications and these are just two examples to prove the point. There can be other meaningful ways of tracking the fuzzing progress. We do not go into detail in these examples because they are of no importance for our further work. They shall just demonstrate that there are different methods to monitor the state and progress of fuzzers.

2.3.4 Conclusion

To summarize the main contents of this section for RQ1, we recall that fuzzing approaches are often extensions of a coverage-based approach. These approaches gradually increase the tested amount of code by visiting new code parts. There are various goals which are harmonized with the respective testing domain [PLS+19]. Information like the execution path length or the amount of equal characters in a comparison are only two examples of possible goals. Typically the goals of fuzzers expand the coverage-driven approach like in Padhye et al. [PLS+19]'s approach but it is also possible to refrain tracking the code coverage and measuring different data during runtime, like for example memory activities, as stated in the previous subsection.

Guiding towards maximizing execution path length, managing checksum and password checks or increasing memory allocation are some examplary goals and were successfully used for fuzz-testing [PLS+19]. We are going to compare our approach to some of the previously introduced approaches as points of reference.

Looking retrospectively at the introduced approaches, the most commonly used goal is maximum code coverage. However, there are different kinds of additional goals to make fuzzers more efficient in specific domains like SlowFuzz or PerfFuzz, Mem [PLS+19], BigFuzz [ZWG+21] and more. Other goals like in 2.3.2 enable the fuzzer to reach parts which were unreachable without input structure knowledge before. Depending on the testing domain, sometimes monitoring the memory or other dynamic analysis methods are reasonable and code coverage is not the information the tester wishes to collect [YRKS22]. Varying testing objectives contribute to our recommendation of analyzing the system which shall get tested and considering your testing intents for deciding how to guide a fuzzer.

At this point, we want to mention that a fuzzer consists of more than just its goals for guidance. Guiding the fuzzer towards better inputs is not only about defining and following domain-goals to determine which inputs should be saved. It is also about their prioritization. Seed prioritization, e.g. favoring more recent or certain seeds, plays a role in the efficiency of a fuzzer, too. A proper definition of how to prioritize, can help the fuzzer to choose favored inputs [LZJ+19]. And handling difficult parts of a system can not only be solved with specific goals. There are different ideas of getting rid of checksum or password checks like for example a fuzzer by Peng et al. [PSP18]. T-Fuzz guides via code coverage and as soon as it fails at increasing code coverage for some time, a tracing algorithm detects the checks which could not be traversed. By removing these checks, new code paths that potentially have bugs can be explored [PSP18]. By altering the SUT and removing certain code parts, false-positive bugs might be detected. However, this is a good alternative to 2.3.2 for reaching code parts which are almost unreachable via normal coverage-based guidance.

2.4 Available Fuzzing Tools

In this section, we recap which tools were already introduced in this chapter and present another interesting tool. We therefore address RQ3 which is about fuzzing tools and the goals they cover. In our context, the testing of SPLs, we examine the role of embedded systems for fuzzing tools. Here, differences and potential difficulties are to be looked at. And we briefly look into a scientific article which lists numerous available fuzzers which address embedded systems.

2.4.1 Open Source Fuzzing Tools

The first fuzzer we introduced, AFL [Zal19], is probably the most widely known open source fuzzer. The other fuzzers we introduced, FuzzFactory [PLS+19], PerfFuzz [LPSS18], SlowFuzz [PZKJ17] are all mutational and open source. The goals covered by these fuzzers are elaborately explained in the previous section. Furthermore, there are many interesting free-to-use fuzzing tools, like for example CI-Fuzz [Cod23], which is very well documented and allows new users to comprehend the fuzzers capabilities easily. Surely, these are not suitable fuzzers for any program but reading the respective articles and testing these few tools, might help a lot in order to better understand fuzzing and the differences between tools. Also, the understanding of basic principles of fuzzing, facilitates the development of your very own fuzzing tool. For large testing tasks, we suggest to always consider developing a tool from scratch asides from choosing and extending available open source fuzzers. Weighing the pros and cons will facilate taking a choice. There might be a better understanding of how to implement new functionalities to improve and adapt the tool to your desires

and SUTs. Free choice of programming language, could also be an argument for an own tool because it might be easier to implement a functioning (or efficient) interface for communication to the SUT. On the other hand, the potentially tremendous time cost could be a reason for an available tool.

2.4.2 Fuzzing Embedded Systems

Before fuzzing embedded systems, there are a couple of decisions to be made. It is generally much more difficult to detect faulty behavior or even crashes in the SUT. The main reason is an access or interpretation problem of the fuzzer due to the closed natures of very hardware-oriented SUTs [Eis22]. Other difficulties can occur due to tough multi-architecture support or too limited resources [YRKS22]. “Simpler” user applications have been studied more extensively and are oftentimes more straight forward to fuzz [YRKS22]. The works of Eisele et al. [EMS+22] and Yun et al. [YRKS22] elaborately examine the difficulties of embedded fuzzing. The investigation of a general, increasingly product-independent fuzzing tool is a very interesting topic and findings might change the state of the art of embedded fuzzing completely. Unfortunately, Eisele et al. [EMS+22] found no easily applicable solution for this purpose. Currently, embedded fuzzing is difficult and in order to test a single embedded system, there are hurdles to be cleared before fuzzing.

Yun et al. [YRKS22] provide detailed information about existing embedded fuzzers. They mention both open- and closed-source. Their work might potentially help in finding a fuzzer that is capable of fuzzing given SPLs but in either case, it might be worth looking at some product-specific solutions mentioned in their paper if you are stuck in your own development of a novel fuzzing tool for an embedded system.

Summary

In this chapter, we introduced fuzzing in general and addressed the two research questions which are directly revolving around this topic: RQ1 and RQ3. We explained what fuzzing stands for and presented commonly used goals, as well as available fuzzing tools.

In the next chapter, we want to specify which variability models there are in the context of SPLE for the first part of our second research question: RQ2.

3 Variability Models in SPLE

In the context of Software Product Line Engineering (SPLE) we need models to illustrate the variability of features and components within product lines. It is neither feasible to instantiate and test all product configurations nor possible to manually model every configuration and consider them for the derivation of test scenarios. Therefore, valid configurations are consolidated by models which show the variety of features and constraint specifications in general.

As an important foundation for this thesis, we devote this chapter to addressing RQ2. We begin with the first part of the question, namely the introduction of different variability models. The questions second part, how they can be used to identify product-specific goals, follows in Chapter 4. We introduce existing variability models that are commonly used when studying the domain of a SPL. In this chapter, we explain the contained information and the structure of different models by analyzing the Body Comfort System (BCS) [LLLS17] and other examples. The introduction of the models and a general understanding therefrom is fundamental for the identification of product-specific goals which follows in the next chapter.

Such a model-based approach is part of domain engineering and gives us information about the system and its domain. We now try to create an understanding of the systems environment based on the information we get from variability models. Within the example of the BCS, we will highlight and clarify interesting model parts that are eminent for the BCS. Even though the variability can be modeled textually, we solely focus on graphical illustration. Graphics generally provide a clearer view and are easier to analyze and understand.

3.1 Feature Model

Feature models are used to graphically illustrate the set of all features that are configurable in a SPL. It does not only illustrate the whole tree of available features, it enables to portray relationships between features by providing symbols for several constraints [Mar06].

The root node corresponds to the general conceptual system and children nodes correspond to features, subparts of the system. A grandchild node would mirror a subfeature. For the definition of a valid configuration, the model offers the connectors shown in Figure 3.1 to illustrate constraints. The feature model can distinguish mandatory and optional features and offers an or- and xor-operator to portray constraints between child nodes. Dotted cross-model arrows with the captions “require” or exclude”, additionally enable the illustration of constraints between features of distinct subtrees [Mar06]. For example a desktop PC would have the mandatory features of a mainboard and a processor among other things. But for a valid, functioning configuration of a PC, a mainboard with the subfeature “LGA1200 socket” would exclude processors from AMD and vice versa. In this section, the BCS shall demonstrate how feature models can model the variability of SPLs sufficiently.

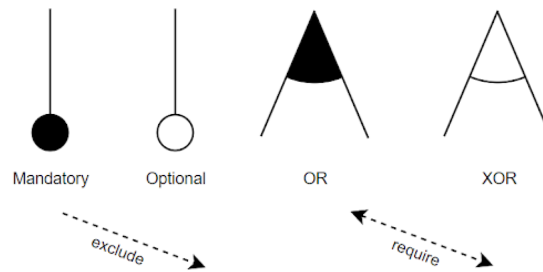

Figure 3.1: The operators of feature models. Summarized from [LLLS17; Mar06].

3.1.1 Usability of Feature Models for Testing

How useful and informative a feature model can depend on the creator of the model. Logically, there is only one correct solution for constrained relationships between features like "require" or "exclude". Therefore, features with such relationships can be considered easily. Features without further constraints are less noticeable. If they are mandatory they are in aspects of variability uninteresting. However, some mandatory features might play focal key roles in the system. Optional features can have various reasons for not being mandatory. Assessing the importance of such features to the complete system can distinguish features for luxury or comfortableness from features that might impact other parts of the system. Subfeatures connected with OR- or XOR-constraints can be interesting to examine.

To find the respective modules or features to be tested in more or less detail, one needs to build a comprehensible and informative model. Understanding the relationships between features and the roles of features to other parts of the system helps in the course of this. Just as in programming, using meaningful names and identifiers is one example of a good model [Kuh10]. Later, we are planning to augment the nodes with specific details, supporting the classification of features in terms of their testing importance. The upcoming subsections introduce the BCS and its feature model.

3.1.2 Overview of the Body Comfort System

Sascha Lity et al. introduced the Body Comfort System in a scientific report to present a novel testing framework [LLLS17]. The original BCS was developed and presented by Müller et al. in a tech report in 2009 [LLLS17; Mül+09] but as Lity et al., we use the expanded SPL-version by Oster et al. [Ost+11]. This architectural model is from the automotive domain and the report includes the systems functionality, variability, numerous testing scenarios and further information.

The main components are a door system, a human machine interface, an alarm system and a central locking system. There are many configuration options for these components and several control units execute the software of these subsystems and communicate with each other. But sensors and actuators are the most important units for communication with the vehicle [LLLS17]. In the following, we examine the possible valid configurations.

3.1.3 Feature Model of the Body Comfort System

The Body Comfort System is a great example of broad variety within a product line. To model the dependencies between features, the aforementioned connectors are used. Figure 3.2 shows the complete feature model of the BCS. We want to demonstrate the structure and reasoning of feature models. The “alternative”-key in the legend corresponds to the XOR-connector in our terminology.

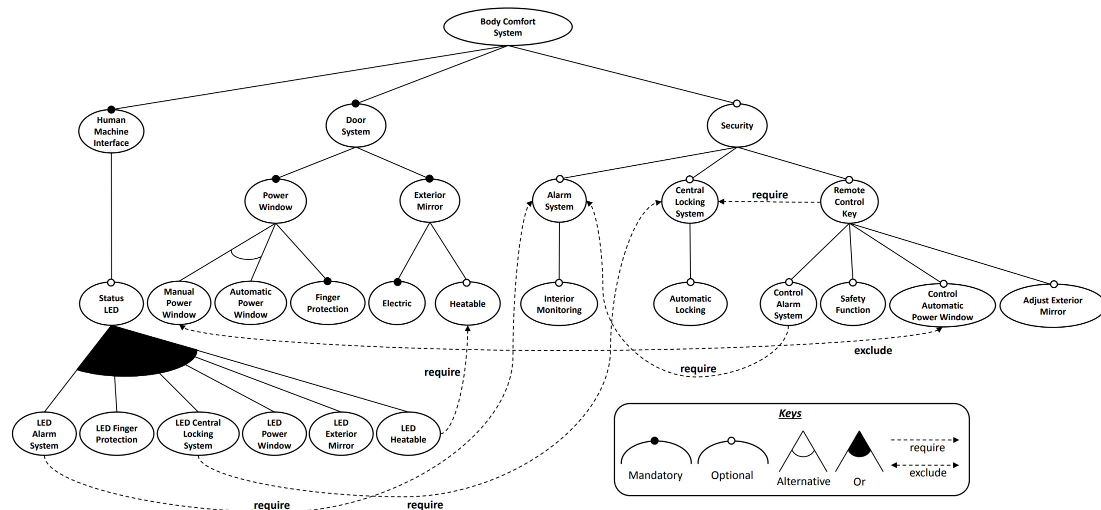

Figure 3.2: The feature model of the BCS by Lity et al. [LLLS17]

The BCS corresponds to the root node and the main components are one level beneath. With our current knowledge, we notice two things: the HMI and door system are mandatory and the alarm system and central locking system are clustered beneath the third feature “security” which is optional as there is a blank circle at the connector’s end.

Human Machine Interface

The first major feature is the human machine interface and it is mandatory. We cannot derive all components the HMI consists of and the feature model does neither model the structure of the HMI nor its input and output signals. For such purposes we need interface specifications and further modeling.

In the HMI, variability exists at the optional subfeature status LEDs: if equipped, the HMI can be augmented with at least one and at most six LEDs, each indicating the current state of other respective components in the car. Naturally, the affected components need to be included as well and are hence connected with require-arrows. Such an arrow is between the LED from the alarm system and the alarm system itself for example.

As the feature model is a variability model, one might think that its only purpose is to specify and model valid configurations, to distinguish valid from invalid configurations. Therefore, there would be no need to include subsystems from a component which do not have any configuration options as these will be similar in all configurations and won’t violate any constraints. But in practice

feature models are helpful to gain an overview of all features, including the mandatory ones without configuration options. Nonetheless, the authors decided to not go into more detail about mandatory subparts of the HMI.

Door System

The second major feature is the door system which is mandatory, too. It comes with a power window and with exterior mirrors. The window is either manual or automatic, visualized by the XOR-connector. The exterior mirrors can be heatable. Contradictory to the assumption that there is no need to model mandatory features that do not have any configuration options, the subfeature "electric" is listed beneath the exterior mirror. Similarly, the window always includes the subfeature "finger protection". While there are no noteworthy mandatory subfeatures of the HMI, the door system has the aforementioned. This shows the leeway in decision-making during the creation of feature models.

Presumably it can be advantageous to include some unconfigurable features in the model which feed the reader with valuable information about the system. In this example, as there is no option for a manual exterior mirror, we might also rename the mandatory "exterior mirror" into "electrical exterior mirror" to eliminate redundant nodes and still include that information. For our thesis, a norm about the inclusion and exclusion of features in the model would simplify the construction of variability models as it would logically be possible to not list the electrical property at all. But generally there is space for interpretation and we will address this in our process of constructing the most suitable variability model for our purpose.

Security

The last major feature is the optional security feature. The alarm system, central locking system and remote control key are the subfeatures and are all optional. Here we see that two features on the same level can not only have a OR- or XOR-constraint but also a require-constraint like between "remote control key" and "central locking system". Naturally, the "control automatic power window" subfeature on the remote excludes the manual power window which is indicated by an exclude-arrow.

3.2 Variability in Space and Time

The terms "Variability in Space" and "Variability in Time" originate from Pohl et al. [PBL05]. They refer to both the multitude of features within one version of a system and the evolution of features over time. In SPLs and especially in SPL testing, the variability of both must be distinguished and modeled. The information acquired from either variability in space or time, can be used to derive meaningful test assets. The following short example shall illustrate the differences:

Consider a washing machine coming with a basic and an advanced version. In 2022, the basic version has no configuration options but the advanced version offers optionally additional washing programs and optionally a touch panel and consists of premium quality materials. In 2023, the manufacturers release a facelifted washing machine "V2". Now both versions come with a revised

touch panel and the basic version still has no configuration options but the advanced version offers a more energy-efficient water pump and can optionally be equipped with Wi-Fi capability or optionally additional improved washing programs.

The overview of the variability in Figure 3.3 coarsely shows the variability in space and in time. The model is inspired by [WKR22]. The difference of material is not of technical relevance in this model, we refer to the software and hardware of the washing machine. The necessity of modeling both variabilities, results from potentially different testing aspects that should be considered.

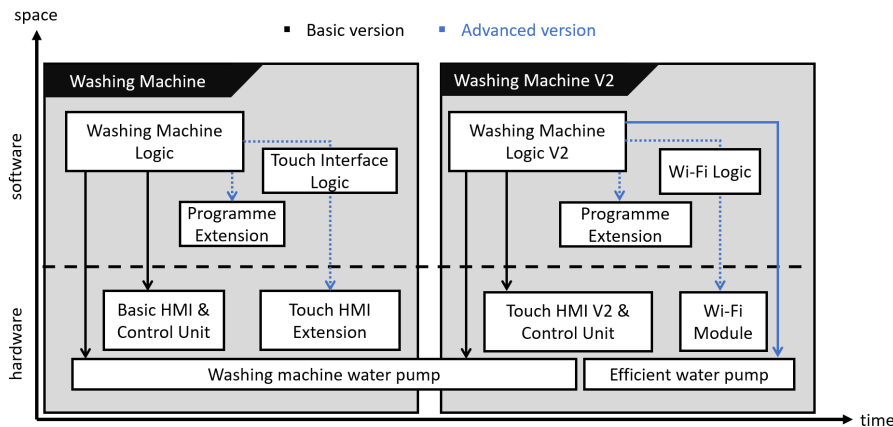

Figure 3.3: The earlier described variability of the fictional washing machine illustrated in a variability model. The syntax is inspired by Wittler et al. [WKR22].

Within this model, improvements and upgrading over time are modeled on the x-axis: the washing machine logic was revised and the separate touch interface logic is probably included in the new version. Also, a new module for Wi-Fi compatibility and new hardware components for efficiency are introduced in V2.

Without knowledge about earlier versions of this system, the tester of V2 might find it unnecessary to pay increased attention towards the Wi-Fi module, due to its little importance to the complete system and would rather test the main functionalities like the touch display or the washing programs. This approach will most probably ensure that the overall system works fine, thus creating general customer satisfaction.

However, by ignoring the variability over time, by not elaborately testing the newly introduced functions like Wi-Fi compatibility, undiscovered bugs might disturb premium customers who chose to include Wi-Fi compatibility for controlling the machine wirelessly. And if the main modules only slightly changed and the first version was successfully tested before, potentially bugs were corrected in the new version. Before releasing V2, of course the main components would need to pass general test cases to rule out unwanted behavior again. Assuming the new version works similarly, we could try to satisfy the premium customers by testing the new Wi-Fi module to its extent.

When designing test cases from the variability of features, most likely test cases for the Wi-Fi module and other optional features would be included. After passing these tests, the overall system is working very well. But we can infer that variability over time can provide us with additional

valuable information to expand or replace test cases that come from variability in space. While designing test assets, it remains an option to discard the idea of focussing on version changes and focus on the current variability of features instead.

In this thesis, we want to stick with analyzing and trading off both variabilities against each other and choosing the best compromise for every scenario. Therefore, we aim to model both the variability in space and time before identifying focal points for guided fuzz-testing.

3.3 More Variability Models

The feature model contains the variability of features, in any case the features which are not predefined completely. It also models the constraints between features, hence modeling only a subset of all possible feature combinations.

We want to examine and focus on varieties of features of SPLs. The process of validating the correctness of configurations is negligible as we do not have to validate the correctness of configurations in this work. Nonetheless, we want to consider constraints to understand the relationships and communication between components.

The following two models are examples of other variability models that can be used to model constraints and features in SPLs:

1. Orthogonal Variability Model (OVM): According to Roos-Frantz [Roo22], the OVM proposed by Pohl et al. [PBL05] distinguishes variation points which correspond to a variable item. And there are variants which correspond to instances. For example connectivity could be a variation point and a variant could be WIFI or NFC. We do not go in any detail here.
2. Decision Model and Notation (DMN): The DMN aims to simplify decision processes by illustrating a variability of features and defining decision rules which depend on the selection of features [Dmn21]. These decisions generally enable the specification of other concerns according to configuration differences in SPLs. By generalization the model can increase the efficiency of decisions.

We want to persue the idea of separating variabilities according to their subject like in OVMs within feature models but we dismiss any syntactic elements from the OVM. The DMNs approach of formulating decision rules might help testers to reason focal points for single instantiations of SPLs. We do not go into detail on other models because we can model the variety of features well-organized with the current feature model already.

3.4 Combined Variability Model

In this chapter we learned about various aspects that need to be considered for the creation of informative variability models. A variability model should be designed so that the desired information is detailed and emphasized. For the application of software testing, we found out that both the variability of features and changes of the system in time should be included in the model.

Following the introduction of the variability models we want to use them to identify product-specific goals. For this, we want to commence the analysis with a variability model that includes the basic information that is indisputably required to express the respective SPL accurately.

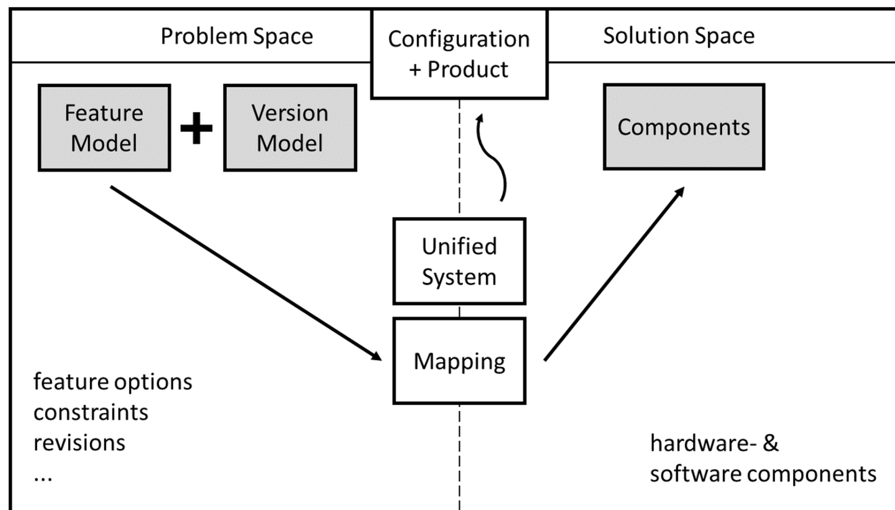

Figure 3.4: Our simplified variability model - the base for product-specific fuzzing optimization. Inspired by the Refined Conceptual Model by Wittler et al. [WKR22].

Figure 3.4 shows the Combined Variability Model, a model inspired by the Refined Conceptual Model by Wittler et al. [WKR22] which is a variability model that differentiates between problem space and solution space. Features and revisions are conceptually included in the problem space. To graphically model the variety of a SPL, we want to use the combination of a feature model and the model from Figure 3.3. In the solution space, all the software and hardware components are modeled. Via a mapping specification, the features can be assigned to components which represent the actual implementation. This will be helpful because we need to examine and understand the relationship of features with peripherals. To separate the essential information for our optimization proposal, we simplified the Refined Conceptual Model into our Combined Variability Model.

Next up, we want to identify product-specific goals from SPL variability models. In the following, we will start the process by analyzing and extending this concept. Naturally, we can specify extensions which improve the model by augmenting it with further details or information about the features and components of the SPL.

Summary

In this chapter, we presented a base for RQ2. We introduced different variability models for SPLs: Firstly, we found the feature model being the most suitable variability model for the variety of features in SPLs. And secondly, we presented a possible design for modelling the variety of features over time. Eventually, we defined the CVM, a model which combines several aspects of the SPL which we further use as a base for deriving product-specific goals for fuzzing.

In the next chapter, we are dealing with the second part of RQ2: how to derive product-specific goals for fuzzing from variability models in order to optimize the fuzzing performance when fuzzing SPL instantiations.

4 Optimization of Fuzzing for a SPL

Guiding the fuzzer towards product-specific goals can make the fuzzing of products more efficient and oftentimes requires little preparation. We want to propose a general process which describes all the steps needed for adapting a fuzzer to a given SPL. When testing an instantiated product of the respective SPL, we aim to achieve better results than unadapted fuzzers which are based on product-independent guiding techniques like code-coverage or execution path length, with the idea of product-specific guidance for fuzzing and the help of augmented variability models. To support our proposal, we apply the process to the earlier introduced example of the BCS in this chapter and evaluate the effectiveness and efficiency in the next chapter after testing an exemplary valid configuration of the BCS in a virtualized environment.

To have a solid starting point for the optimization, we chose the Combined Variability Model (CVM) as a model which contains suitable and extensive information about variability. Eventually, that means there has to be sufficient information about all components mentioned in the CVM, in order to apply each stage of the process. In the following, we want to start by focusing on the feature model.

4.1 Creation of a complete Feature Model

The main source of information we need about SPLs is about the features and their constraints. However, the validation of configurations is not part of this work, resulting in a more indirect need of information about constraints for us. In the first step of our process, we suggest creating a reasonable and complete graphical feature model which illustrates all features and constraints of the SPL.

Possibly there already is an existing feature model which can be reviewed. If that is not the case, we need to agglomerate the required information about the variety of features and constraints between them by analyzing the available sources of information. These could be textual descriptions or graphical variability models that might follow their own syntactic guidelines.

The feature model should have a reasonable structure. A badly thought-out feature model could have all the mandatory features on the same graphical level even though some of these features might be of significantly less importance than others or might be subfeatures of others. In the best case scenario, the feature model is neatly arranged and easy to comprehend.

In our running example of the BCS, we already have a feature model, provided by Lity et al. [LLLS17]. For illustrative reasons, we slightly adjusted the provided model, in order to improve the ability to apply all the steps to the model. The following Figure 4.1 shows an example of all physically cohesive features and subfeatures being close together with the subfeatures beneath their parent features. All the constraints are modeled. The feature model meets our expectations for this step.

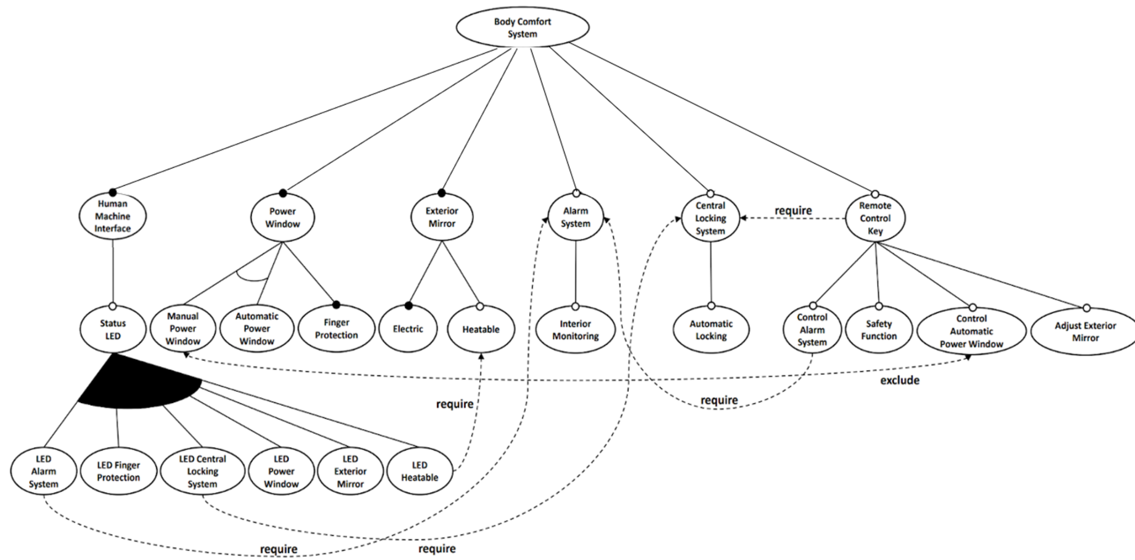

Figure 4.1: A simple yet complete feature model of the BCS. Adapted from [LLLS17].

4.2 Augmentation of the Feature Model with Categories

We aim to create a better sorting of all the features in this step of augmentation. If the first step was done deliberately, for example features that share properties or belong to a family of similar features are sorted in some way and are presumably graphically close to each other. To emphasize clusters like these in our feature model, we want to define actual categories and accumulate features underneath the categories. We distinguish between two types of categories: inheritance- and testing categories.

4.2.1 Inheritance Categories

The main aspects of these categories are similarity and affinity. As the name implies, inheritance categories are thought to be like inheritance in programming languages like Java. In order to keep the amount of categories manageable, features that would arguably belong to the same inheritance tree, might belong to the same inheritance category in our understanding nonetheless. If one feature has no features to build a logical inheritance category in the original sense of inheritance, before defining a category for this single feature only, it might be reasonable to try finding a suitable category. We extend the idea of inheritance similar to how interfaces work in Java. For such cases it needs to be decided whether the combination is rational or if separate categories are required.

In the BCS, we found three reasonable inheritance categories: the Human Interface (HI), the Door System (DS) and the Security System (SEC). The augmented feature model is shown in Figure 4.2.

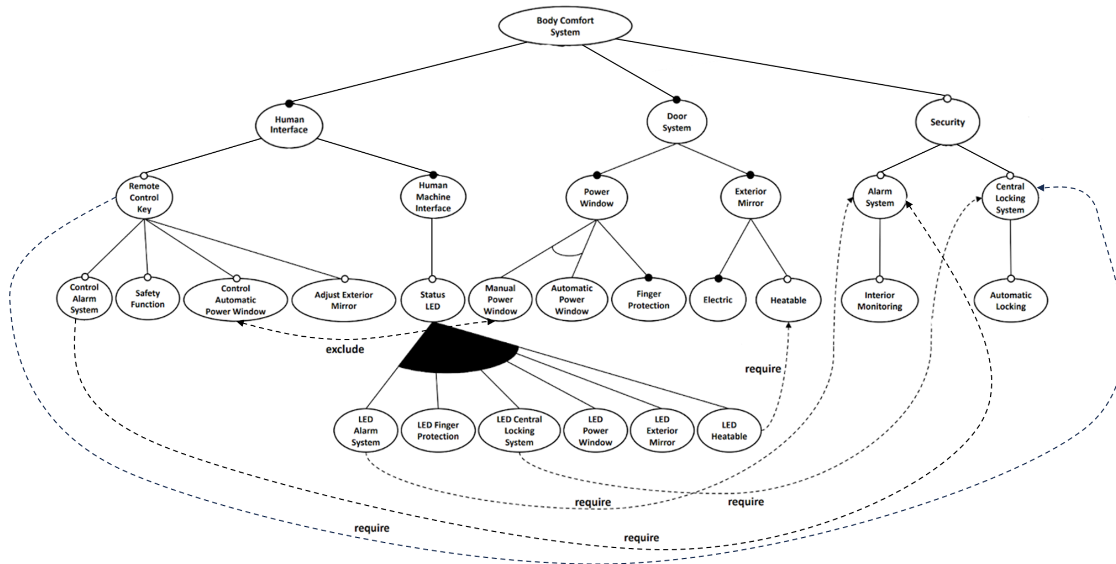

Figure 4.2: Optimizied clustering and categorization in the feature model of the BCS. Adapted from [LLLS17].

4.2.2 Testing Categories

With testing categories we want to allow predefined testing priorities for a given product line. There can be different parts of a system, which could be recognized as essential testing targets.

While testing a car, it seems very reasonable to search for safety-critical bugs and errors. We defined the category Safety (SAF) and highlighted all safety-critical features from the feature model. However, there are many more possibilities for testing categories such as mandatory features, time-critical features and so on. Figure 4.3 shows one possible way to include testing categories and highlight their features in the feature model.

4.3 Augmentation of the Feature Model with Variability in Time

In this last step of augmentation, we want to analyze the information about versions, updates, and new releases of the system. The goal is to include the information in a trimmed way in the feature model. A good initial situation would be if we already have a graphical version control model which illustrates the history of the respective SPL. If that is not the case and the only available source of information is textual, we suggest focussing on the release that is to be tested.

A graphical model of variability in time could look like the fictional model in Figure 4.4. It is just an examplary extract and not the whole amount of features of the BCS are included. Here, the differences between the newest release and the release before that, are the only changes that are illustrated. For our optimization process a similar model or the equivalent level of detail suffices.

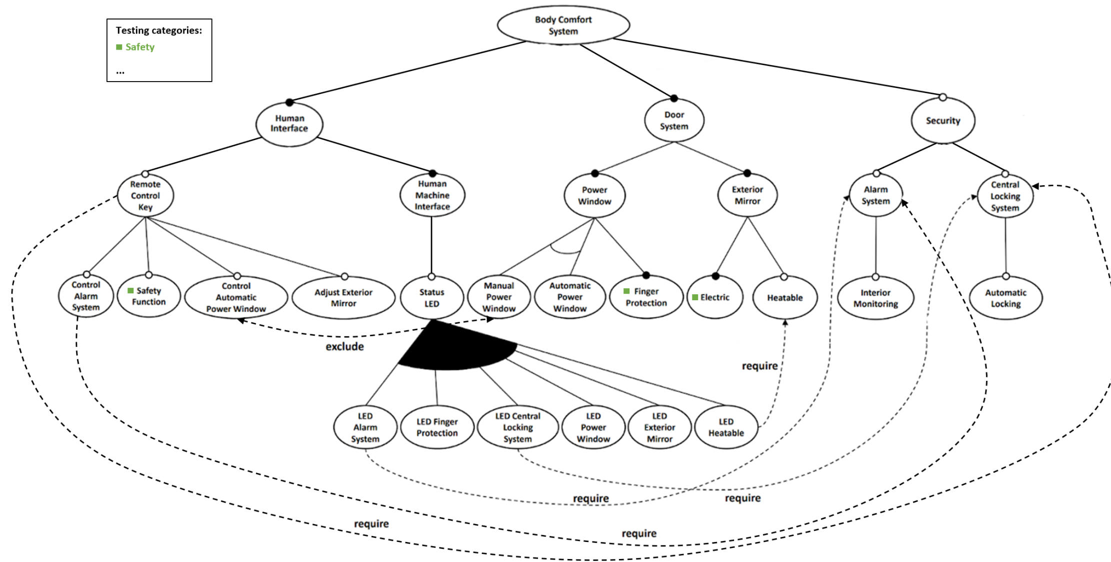

Figure 4.3: Determination of testing categories in order to guide the testing specifically. Adapted from [LLLS17].

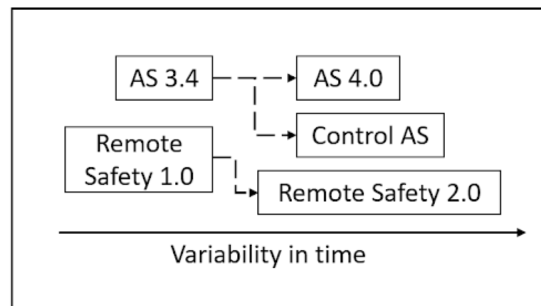

Figure 4.4: An invented short example of potential variability in the BCS.

For an overall improved overview, it is convenient to highlight the recently updated features and the newly added features in the feature model. We propose highlighting with colored icons such as in Figure 4.5. At this point, the feature model should have a proper structure and level of detail and should include all the relevant information for the further steps.

4.4 Assignment of Priorities to the Features and their respective Components

The previous steps can be done prior to the existence of an actual product configuration of the SPL with just the information about the variability of the SPL. From now on, the other parts of the CVM are essential, too. The instantiated product, with all equipped software and hardware components and information about where features are in the project files, is required. In our case, we simulate the hardware components virtually via classes which contain the signals from hardware sensors

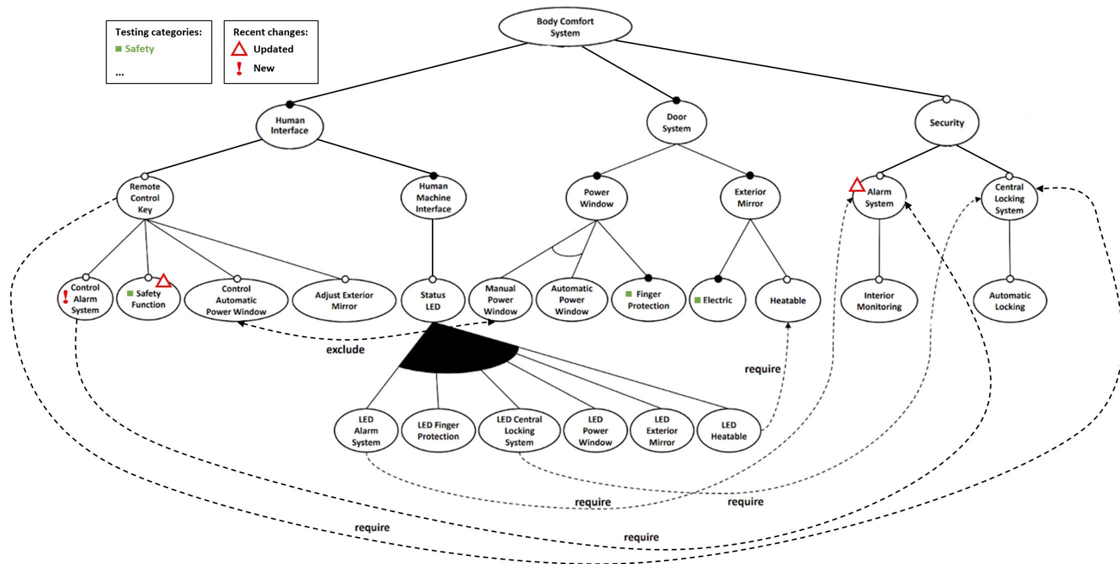

Figure 4.5: The highlighting of updated and new features in the feature model. Adapted from [LLLS17].

and the signals for actuators. With an adequate fuzzer, it can also be possible to choose other ways of fuzzing without a complete simulation of the product but in order to support our theory, we constrain to the easy comprehensible simulated example of the BCS. Further information about fuzzing embedded systems can be found in the scientific papers cited in Section 2.4.2 for example.

With the augmented feature model and the information about the mapping, we aim to create a compact and informative table which includes priorities for every feature. The idea behind these priorities without knowledge about the fuzzer to be used, is that features with higher priority values will direct the fuzzing process, hence the saving and the election of inputs, in their favor. That means features with higher priorities are explored earlier and to a larger extent than features with lower priorities.

Regarding the BCS, the following tables show the priorities for every component of each inheritance category that we defined earlier. The first column lists the features, the second column contains the mapping information about the components where the respective features are implemented. Basically these two columns standardize how the mapping is illustrated but do not include any new information. The third column contains the priorities for each feature. The goal is to fill the third column with reasonable values, by analyzing the feature model and noticing important focal points. Recently updated features should have increased priorities and completely new features should have comparably high values. This step is not indisputable and we suggest discussing this process in a team to achieve a good distribution of priorities for the specific product. For example, all members could independently fill the third column in a first step and fill it again as a team in a second step. Note that testing categories do not need another redundant table because all features are contained in their respective inheritance categories.

HI-Feature	**Component**	**Priority**
Human Machine Interface	HMI.java	1
Status LED	LED_*.java	1
LED Alarm System	LED_AS.java	5
LED Finger Protection	LED_FP.java	3
LED Central Locking System	LED_CLS.java	5
LED Power Window	LED_PW.java	3
LED Exterior Mirror	LED_EM.java	1
Remote Control Key	RCK_CTRL.java	1
Control Alarm System	-	-
Safety Function	RCK_CTRL.java: door_open_change() rck_but_lock_change() rck_but_unlock_change()	5

Table 4.1: The products mapping of all existent features within the HI-category augmented with reasonable priorities.

DS-Feature	**Component**	**Priority**
Power Window	PW.java	6
Manual Power Window	-	-
Finger Protection	FP.java	10
Exterior Mirror	EM.java	5
Electric	-	-

Table 4.2: The products mapping of all existent features within the DS-category augmented with reasonable priorities.

SEC-Feature	Component	Priority
Alarm System	AS.java	10
Interior Monitoring	AS.java: im_alarm_detected_change()	10
Central Locking System	CLS.java	6
Automatic Locking	CLS.java: car_drives_change()	6

Table 4.3: The products mapping of all existent features within the SEC-category augmented with reasonable priorities.

4.5 Methodology of our Approach

In order to guide a fuzzer product-specifically, we defined categories and assigned priorities to every feature. Our goal is to favor more important features in the product during fuzzing. The methodology is inspired by SlowFuzz [PZKJ17] and PerfFuzz [LPSS18].

Generally speaking, we guide the fuzzer by maximizing the execution path lengths separately for each of the earlier defined categories. More exactly, we do not measure the execution path length in just one dimension such as in SlowFuzz and we refrain from measuring separate execution counts for every single program location such as in PerfFuzz. An input is saved to the set of seeds whenever some counter increases. By dividing the project into categories it is possible tracking the progress of the categories independently and getting stuck in local optima like SlowFuzz is less probable because with multiple seeds, optimizing the execution path lengths of different categories, new parts in a locally optimized category can be explored, thus escaping the local optima due to seed variety.

To enable more specifically directed fuzzing, another enhancement is that the visit of a basic block does not necessarily increase the respective counter by one. With priorities for the features and the mapping from feature to program parts, every basic block is augmented with instrumentation which increases the counter of the category where the basic block is located. It adds the priority value of the most specific mapping to the respective counter. For features that are in an inheritance- and in one or more testing categories, the counters of all categories in which the feature is included are increased by the same value.

A very important part of fuzzing is the election of a seed which will be mutated for the upcoming round. Initially, all seeds have the same election probability. Our approach suggests assigning a probability to every newly added input according to the formula:

$$P(\text{category}) = a + \text{categoryPrio} + b \times f(\text{increase}) \quad \text{with: } f(\text{increase}) = 1 - 2^{-c \times \text{increase}}$$

Then, $P(\text{seed}) = \max\{P(\text{category}) \mid \text{all categories}\}$ is the resulting probability. We chose this formula in order to allow some scope of variability with the parameters a, b, c. categoryPrio is 0.0 by default but categories can receive higher values up to 0.4 theoretically, so that inputs which improve the execution path length in the respective category, have an increased probability to be chosen. 0.4 is the highest possible value because otherwise there could occur probabilities greater

than 1. We chose $f(\text{increase})$ to be an exponential function since it returns 0 if the execution path length did not increase and converges to 1 if the increase goes towards infinity. Consider the following for choosing a, b and c:

1. $a, b, c > 0$
2. $a + b = 0.5$
3. the greater a, the less significant is the amount of increase
4. probabilities of small and great increases get increasingly more equal with a greater c

Logically, the sum of all seed probabilities should be equal to 1 because we want one seed to be chosen in every round of fuzzing. Therefore, the other probabilities are reduced by $\frac{P(\text{seed})}{|\text{seeds}|}$ when a new seed is added. To ensure a minimum probability for every seed in the set of seeds, we determined that the probability of every seed must be greater than or equal to $\frac{1}{10 \times |\text{seeds}|}$ at any time. If the reduction would result in less than $\frac{1}{10 \times |\text{seeds}|}$ probability for one seed, $\frac{1}{10 \times |\text{seeds}|}$ is assigned as the new probability and the rest is subtracted equally from the other remaining seeds in order to remain the aggregate probability of 1.

Another idea for making the fuzzing process more efficient is getting rid of older seeds that do not cause input diversity. That means if we add a new input, we might delete existing seeds that achieved worse execution path lengths than the new input in every category if the contents are very similar. How the checking of similarity can be implemented will not be elucidated in this work but depending on the level of detail about the input format there are different possibilities. Without any knowledge about the format, for example the hamming distance [TK09] in combination with a threshold value could be calculated to sort out resembling seeds which performed worse in all categories. With knowledge about input format, the presence of inverted data at booleans or numbers, huge differences between numbers or the presence of extreme values could be an argument to not delete a worse performing older seed. This filtering of old inputs, allows the more interesting and potentially unexplored seeds to be chosen with higher probability.

Our proposal revolves around optimized guidance and election of seeds. We do not present changes or restrictions to the forms and methods of applicable mutations of the fuzzer. As explained in Chapter 2, there are various kinds of mutation methods and a good amount of variety and randomness is what we presume.

4.6 Insertion of Instrumentation to collect Runtime Data

In theory, everything to start augmenting the selected fuzzer for testing the desired SPL is now prepared. The fuzzer requires integer variables for each category which contain the current maximum execution path lengths. It is reasonable to implement separate sets for storing the inputs that result in crashes and hangs like in Algorithm 2.1. It is a matter of choice whether the fuzzer should run until it is stopped manually, until the set of seeds stays constant for a given amount of time or simply until the first finding.

In order to realize the idea of deleting similar seeds, we need to store objects in the set of seeds that contain the actual input data on the one hand and the execution path counts on the other hand for eventual comparisons.

The following figure shows how the instrumentation of the product under test could look like, inspired by the syntax of FuzzFactory [PLS+19]:

Domain d: K = {HMI, DOOR, SEC, SAF}, V = ℕ, A = ℕ, a_0 = 0, a ⊳ v = max(a, v)	
Hook	**Instrumentation**
entry_point()	***insert_after***('∀m ϵ {HMI, DOOR, SEC, SAF}: dsf(m) ← 0')
new_basic_block()	k ← current_program_loc() m ← categoryOf(k) n ← priorityOf(k) ***insert_after***('dsf(m) ← dsf(m) + n')

Figure 4.6: Required instrumention for our approach in the syntax of Padhye et al. [PLS+19]

Every time a basic code block is visited, the respective counter(s) are incremented and if the execution path of at least one category increases, the internal counters that store the maxima are updated and the input is saved to the set of seeds.

With white-box knowledge and therefore access to the complete code, the insertion of instrumentation will be straightforward and a compiler could be developed which distinguishes program locations and instruments as specified in the mapping. If the code of the product is sealed and the fuzzing is supposed to be done with gray-box knowledge only, a compiler that reads the priority tables including the mapping information which generates appropriate instrumentation and traceable program locations without knowing the size of the actual code has to be developed. Technically, the black-box information, therefore no information about the code within the method, should suffice to define how the basic blocks in the respective method would need to be instrumented but potentially there might be limits, depending on the format of the product that is to be tested. However, we do not want to examine the development and limits of an automated compiler in order to evaluate our proposal. In order to evaluate our proposal, the example white-box project of the BCS, is a simple yet effective way to compare the fuzzing results of unimproved fuzzers and our proposal.

Summary

In this chapter, we defined a process of optimizing fuzzing for SPLs. We propose an extended version of an execution path length guided fuzzer which utilizes product-specific categories and priorities. With this proposal, we present our own solution for RQ2 of how to identify product-specific goals from variabilty models.

In the next chapter, we want to apply the process and evaluate the performance of our solution. By applying the theoretical steps of the process in this chapter already, we could use the figures to explain more simply with examples. The following chapter addresses the steps regarding the implementation and execution.

5 Evaluation

In the previous Chapter 4, we explained the entire optimization process. For any sufficiently documented SPL, we propose a stepwise process describing the optimization of a suitable fuzzer. In this chapter, we describe the technical realization and the evaluation of the results, in order to evaluate the effectiveness of the new fuzzing strategy and to prove the extent of thereby originating benefits.

5.1 Realization of the Fuzzing Project in Java

We are applying the process and the fuzzing to the BCS, due to its manageability and the very accurate documentation by Lity et al. [LLLS17]. With the help of an architecture model out of their case study [LLLS17] which includes all inputs and relationships between the components , we chose to instantiate a product accordingly in Java [Aus23].

5.1.1 The BCS instantiation

The exemplary instantiation has not every single possible feature configured: it misses the Heatable, thus the Heatable-LED and has the Manual Power Window instead of the Automatic Power Window. There is no specific reason, it is just a design decision and other valid configurations could be tested similarly. Just like in the mapping in Table 4.1, Table 4.2 and Table 4.3, every major feature corresponds to their own class and minor features make up parts of classes. Certainly, the product's hardware components are simulated by respective classes, too. The input signals from sensors or interfaces towards components are contained therein. In sum, all these signals make up the input file for fuzzing. Files of this structure are then mutated in every round of fuzzing. The following lists all signals:

- `int pw_but_mv_dn`
- `int pw_but_mv_up`
- `int em_but_mv_left`
- `int em_but_mv_right`
- `int em_but_mv_up`
- `int em_but_mv_dn`
- `boolean deactivate_as`
- `boolean activate_as`

- `boolean confirm_alarm`
- `boolean release_pw_but_up`
- `boolean release_pw_but_dn`
- `int em_pos_left`
- `int em_pos_right`
- `int em_pos_top`
- `int em_pos_bottom`
- `int pw_pos_up`
- `int pw_pos_dn`
- `int door_open`
- `int time_rck_sf_elapsed`
- `boolean rck_but_lock`
- `boolean rck_but_unlock`
- `int pw_rm_up`
- `int pw_rm_dn`
- `int car_drives`
- `boolean finger_detected`
- `boolean key_pos_lock`
- `boolean key_pos_unlock`
- `int time_alarm_elapsed`
- `int as_alarm_detected`
- `int im_alarm_detected`

5.1.2 Bugs in different Components

For testing purposes, six of the components are faulty in this product instantiation. We examine the fuzzers' abilities in finding these distinct bugs. They activate in various ways and we refer to the bugs under the following names:

- **Mirror-Bug**: A bug which doesn't allow the simultaneous illumination of more than one LED indicating the current position.
- **Window-Bug**: A bug which occurs in a certain state when the remaining space to the top of the window doesn't match the current state of the window.
- **Remote-Bug**: A bug which sometimes ignores the pressing of the unlocking-button.

- **LockingSystem-Bug**: A bug which unlocks the car under a certain combination of inputs even though no signal initiates the unlocking.
- **FingerProtection-Bug**: A bug in which the protection system doesn't activate after detecting a finger because the state of the window indicates that the finger is not clamped. It should activate nonetheless due to potentially delayed system information.
- **AlarmSystem-Bug**: A bug in which the alarm system is not shut off if the elapsed time is higher than 110 seconds. Normally, it would be deactivated after at most 100 seconds.

5.1.3 The domain-independent Fuzzers

We want to compare the performance of our approach to several non-optimized approaches. To create similar starting conditions, all fuzzers were created from scratch in Java, according to their respective ideas. In this way, the starting conditions are equal in terms of compatibility to the product, the set of seeds and the mutation variants Because all fuzzers utilize the exact same variants of mutation for their seeds. With the white-box knowledge we have, we defined reasonable mutations: the inversion of booleans, the increment, decrement of integers, as well as the setting to extreme values like the smallest or biggest number, 0, or other predefined values. This is a subset of similar mutation methods which are presented in the first chapter of the AFL documentation by Zalewski [Zal19].

The following three fuzzers are the points of reference for the comparison:

1. **LengthFuzz**: a fuzzer which tracks the execution path length and saves an input if it increases the length. Hence, the key idea corresponds to SlowFuzz [PZKJ17].
2. **CoverageFuzz**: a coverage-based fuzzer which distinguishes all basic blocks and saves an input as soon as a new basic block is visited. Multiple block visits of the same block are not a criteria for saving the input.
3. **AdvancedCoverageFuzz**: a coverage-based fuzzer just like CoverageFuzz but it also saves inputs which increase the amount of visits for already visited blocks.

The class “Findingstracker” is responsible for tracking and saving the instrumentation feedback during each round of fuzzing. As soon as the round is over, the current fuzzer updates its internal variables by interpreting the newly acquired information.

5.1.4 PrioFuzz - a Fuzzer according to our proposed Approach

The methodology corresponds to our description in Section 4.5. “Findingstracker” increments internal category counters in each round according to the basic blocks that are visited and the location of the basic blocks. The instrumentation of the blocks follows the priority definitions from the previously defined tables in Section 4.4.

Each input file in the set of seeds is extended by an election probability which is calculated according to the formula $P(\text{category}) = a + \text{categoryPrio} + b \times (1 - 2^{-c \times \text{increase}})$. We are evaluating two combinations of a, b and c.

After adding a new seed, we follow our idea of deleting very similar seeds which is also explained in Section 4.5. By definition, we say that similar seeds feature no unlike signs. That means all fuzzed booleans are equal and all fuzzed integers share the same sign. Such a seed which achieves worse execution path lengths in every category along with it, is not of benefit and is hence deleted from the set of seeds.

We compare PrioFuzz with two different combinations of a and b to the three other fuzzers. We are also examing the role of c and the importance of deleting similar seeds. As we want to compare the impact of a and b on the seed probabilities, we compare the following two combinations of parameters in this work. The numbers were chosen so that one fuzzer has a great b and the other one a comparably small b.

- **PrioFuzz I**: $a = 0.1$, $b = 0.4$ and $c = 0.3$. The probability of new seeds is strongly affected by the different extents of increases due to the greater b.
- **PrioFuzz II**: $a = 0.3$, $b = 0.2$ and $c = 0.3$. The probability of new seeds is less affected by the different extents of increases due to the lower b.

Throughout all comparisons, the parameters of PrioFuzz I and II do not change. Note that we do not modify the categoryPrio values. We specified once in the beginning, that features of the SAF-category are the most important. The HI is least important in our opinion. Therefore, we utilized 0.0 for HI, 0.05 for DS and SEC and 0.3 for SAF. This is subjective and dependent on the intention of the tester.

5.2 Comparison Criteria

We compare PrioFuzz I and II which are approaches based on execution path length, optimized for the BCS, to three other fuzzing approaches. LengthFuzz is execution path-based, too and CoverageFuzz and AdvancedCoverageFuzz are coverage-based approaches. For three different versions of the initial set of seeds, each fuzzer runs *200* times and we calculate the *minimum-, average-* and *median* count of mutations needed to find each of the bugs. In addition, we compare how often the fuzzers were *unable to find a bug* at all. A run stops after *100000* mutations at the latest. The three sets of seeds are of distinct size and the contained seeds feature random and extreme values. The first set contains only two seeds and most values are 0 or false. The second set contains three seeds and has many extreme values. The last and third set is a bigger set as it contains eight pseudo-random seeds.

In this set up, it is not possible to receive and process several input signals simultaneously. Therefore, we defined test cases or more exactly an ordered sequence in which actions are performed or external signals are sent. The product handles the input signals in the order of the sequence. Empirically observable, not any sequence can reach any program part in similar states. Therefore, we are differentiating two scenarios:

- the fuzzers utilize a constant sequence which can theoretically trigger all six bugs that are contained in the product.
- the fuzzers generate a random sequence before each execution of fuzzing and are not able to find some bugs during some execution rounds.

In this chapter, we focus on evaluating the results of fuzzing with a constant sequence. However, we want to prevent any conclusions that might only apply to this single sequence and are therefore comparing the results of both scenarios. The evaluation of fuzzing with random sequences follows in the Appendix A.1.

5.3 Fuzzing Results under a constant Input Order

One possible order of fuzzing the thirty input signals is (30, 18, 6, 11, 19, 17, 13, 21, 24, 7, 22, 2, 20, 10, 15, 23, 3, 1, 12, 5, 25, 14, 29, 16, 27, 4, 26, 9, 8, 28). This sequence includes scenarios in which all six bugs could be triggered with correct input data. There are sequences in which some bugs can never occur because the signals are not arriving in the moment they are expected or in which they would be required to arrive for the bug to occur. We are aiming to compare the minimum-, average- and median count of mutations, as well as the !found-amount, between three different variants of initial sets of seeds. The exact sets of seeds are documented in the Java project [Aus23].

5.3.1 First Set of Seeds

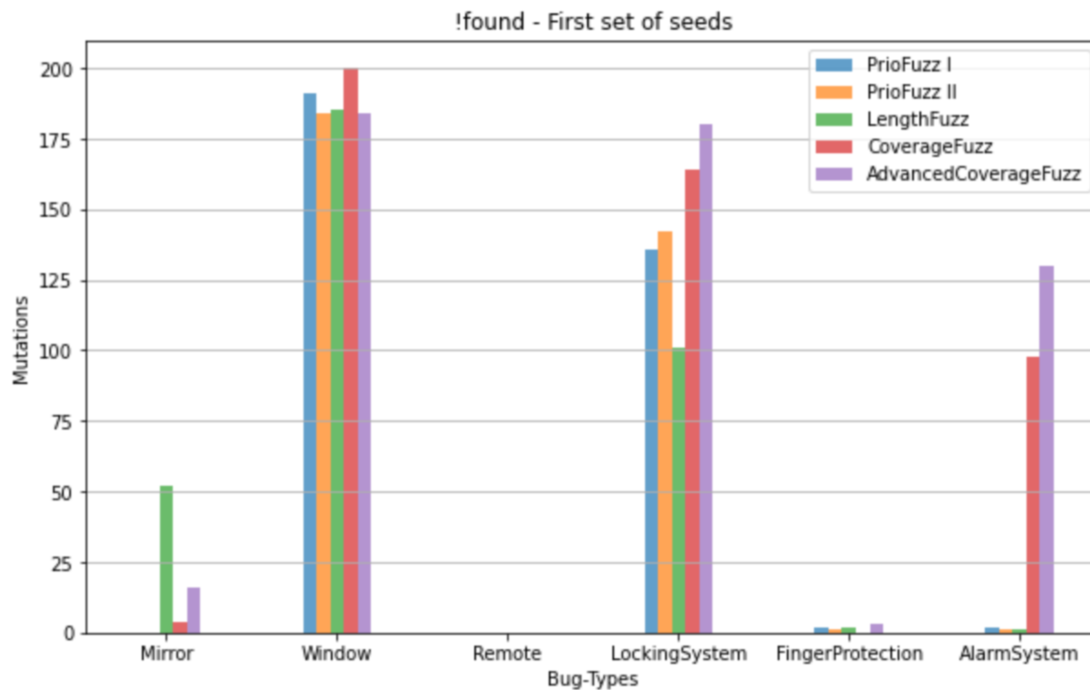

Figure 5.1: First set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

The different nature of the bugs results in differences between all records of them. Figure 5.1 indicates that the Remote-Bug and FingerProtection-Bug were found in almost every execution by all fuzzers and that the Window-Bug and LockingSystem-Bug were most often not found in less than 100000 mutations. CoverageFuzz could not find the Window-Bug in any of the 200 executions.

There are no remarkable difference between PrioFuzz I and PrioFuzz II. Worth mentioning is that LengthFuzz performed better than all other approaches at the LockingSystem-Bug and that the two coverage-based approaches CoverageFuzz and AdvancedCoverageFuzz performed much worse than the other approaches in finding the AlarmSystem-Bug.

While LengthFuzz performed worse than the coverage-based approaches at the Mirror-Bug, the coverage-based approaches performed worse than LengthFuzz at the AlarmSystem-Bug. Overall, the PrioFuzzers were able to consistently keep up with the other fuzzers capabilities of finding all kinds of bugs.

Figure 5.2 indicates that the minimum amounts needed to find the bugs are mostly equal. The Mirror-, Remote- and FingerProtection-Bug can be found with less than 500 mutations by any fuzzer. PrioFuzz I, PrioFuzz II and LengthFuzz also found the LockingSystem- and AlarmSystem-Bug in less than 500 mutations. However, with over 9000 mutations required, LengthFuzz couldn't find the Window-Bug as fast as the other fuzzers. For very small mutation counts, there might be no visible bars even though the value is not exactly zero. CoverageFuzz never found the Window-Bug as can be seen in Figure 5.1. Therefore, there is no bar in any of the Figures 5.2, 5.3 and 5.4 for CoverageFuzz at the Window-Bug. However, for example PrioFuzz I did find the FingerProtection-Bug in a minimum amount of 7 mutations. Still we can not see a bar at the FingerProtection-Bug for PrioFuzz I. Especially for larger y-axis scales, one has to check whether a fuzzer did not find a bug at all in the respective !found-diagram, because that would mean there is no bar in the other diagrams. If the fuzzer did find a bug but there is still no visible bar, that just means that very few mutations were needed to find the bug.

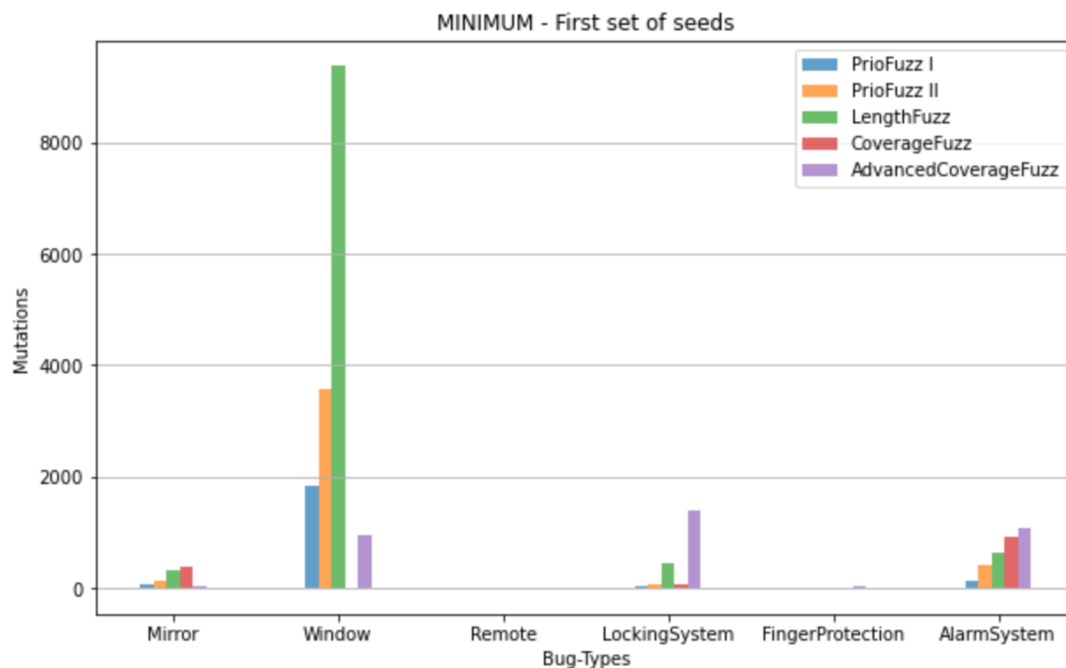

Figure 5.2: First set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

As Figure 5.3 and Figure 5.4 show, both PrioFuzzers achieve great results on average. They perform about twice as good as the second best fuzzer in finding the Mirror- and AlarmSystem-Bug. The only bug were another fuzzer achieved better results, is LengthFuzz's median of finding the WindowBug. However, LengthFuzz was much worse than all other fuzzers in finding the LockingSystem-, FingerProtection- and AlarmSystem-Bug on average. Compared to the worst-performing fuzzers, both PrioFuzzers produce up to ten times better results.

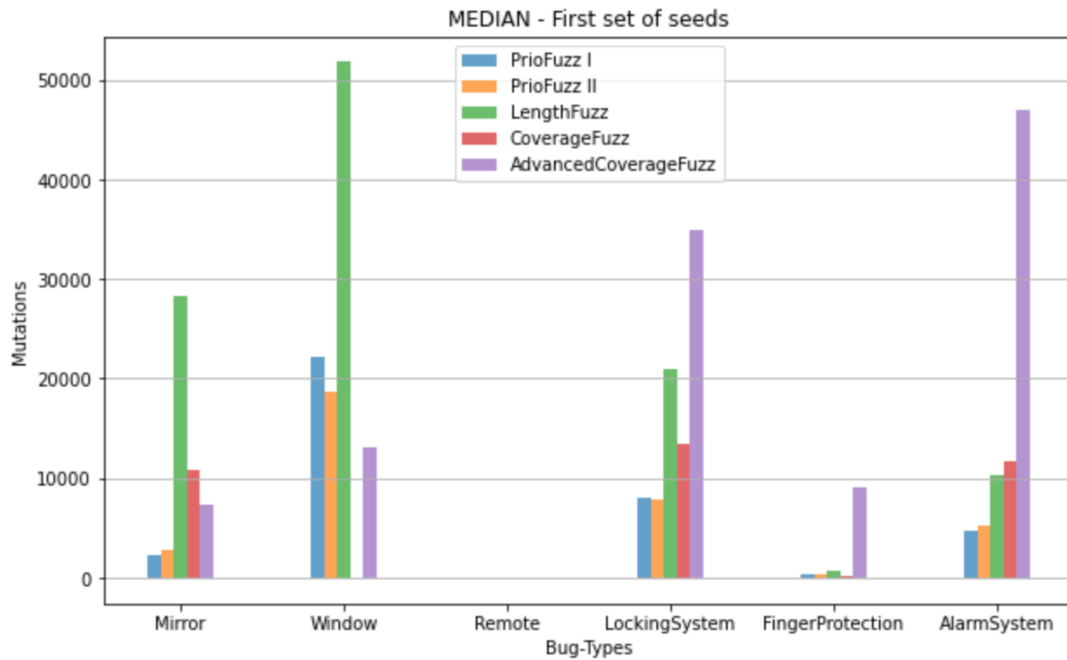

Figure 5.3: First set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

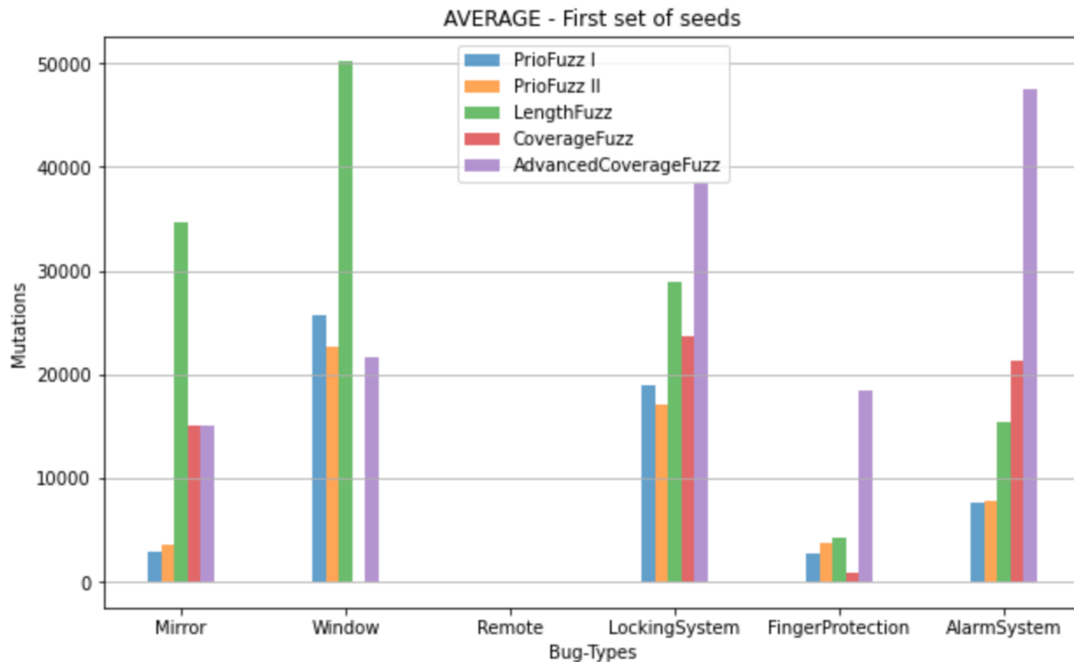

Figure 5.4: First set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

5.3.2 Second Set of Seeds

In this section, we present the results of the fuzzers while utilizing the second set of seeds. Figure 5.5 shows the amounts how often the bugs were found. Comparing Figure 5.1 and Figure 5.5 shows that the Window-Bug and LockingSystem-Bug were even more difficult to find with this set of seeds. The Window-Bug was only found one time each by PrioFuzz I, PrioFuzz II and LengthFuzz. The coverage-guided fuzzers could not discover this bug at all. All fuzzers found the Mirror-, Remote-, FingerProtection- and AlarmSystem-Bug in close to every execution.

Figure 5.6 shows that there are no significant differences between the fuzzers in any bug besides the Window-Bug. But since this bug could not be discovered more than once by any fuzzer, the difference in performance is most probably coincidence.

One detail is eye-catching when viewing Figures 5.7 and 5.8 which show the median and average amounts of mutations required: the CoverageFuzzer features the best values by far. That is most probably a statistical outlier because the averages are calculated from only two occasions in which the fuzzer could find the bug. Besides that, the fuzzers performed mostly similar. The execution path-based approaches - the PrioFuzzers and LengthFuzz, performed slightly better than the coverage-based approaches on average.

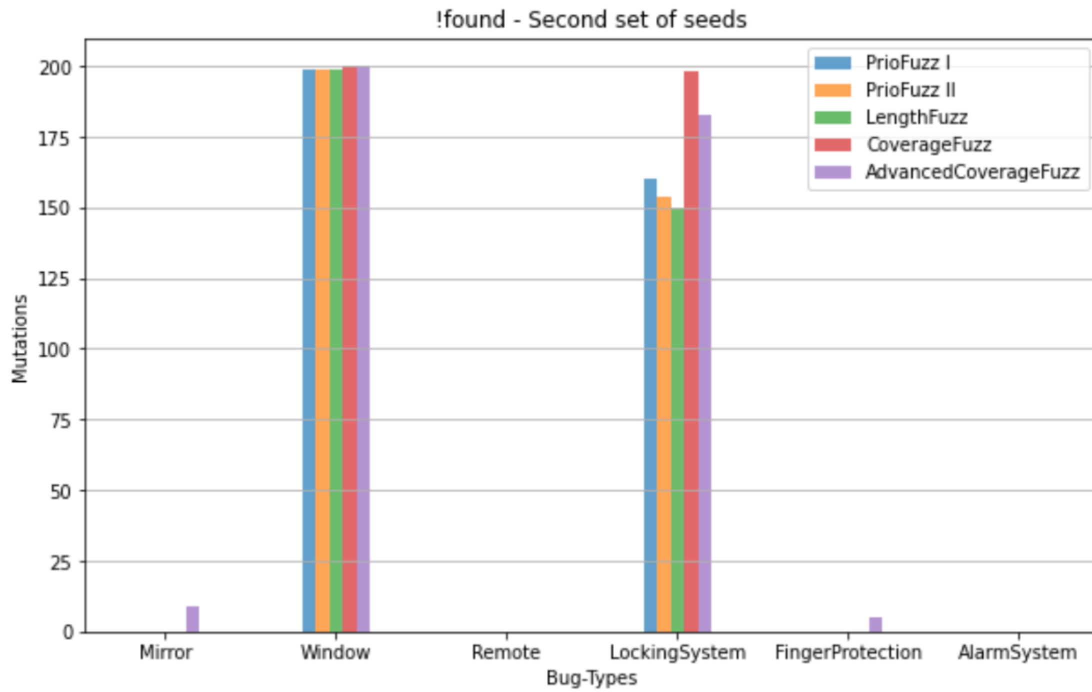

Figure 5.5: Second set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

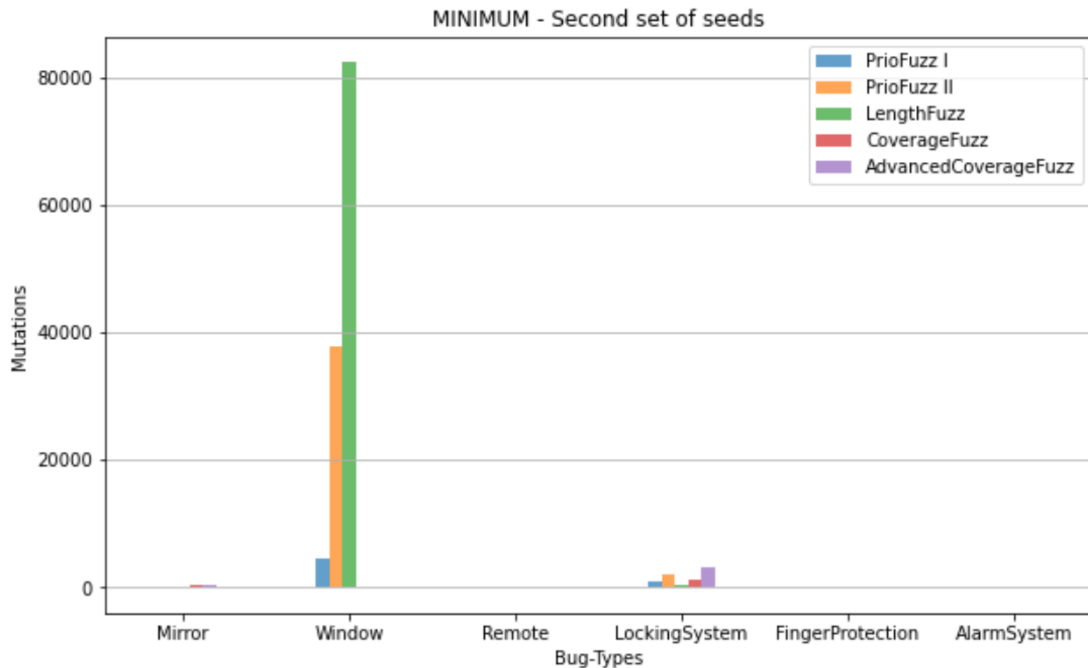

Figure 5.6: Second set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

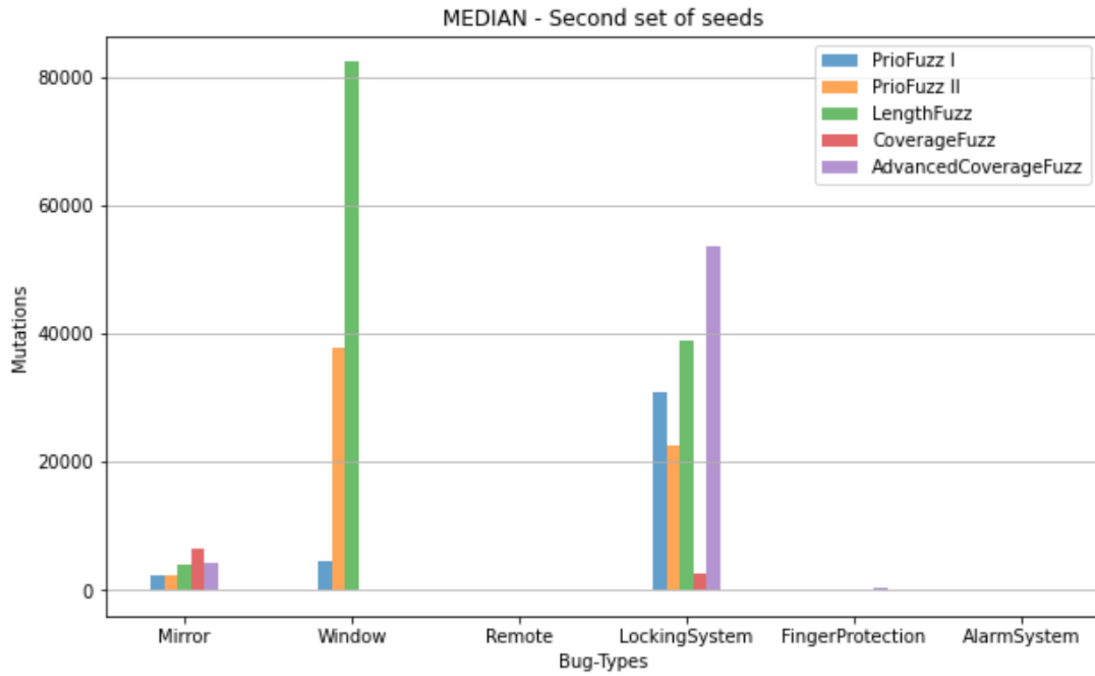

Figure 5.7: Second set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

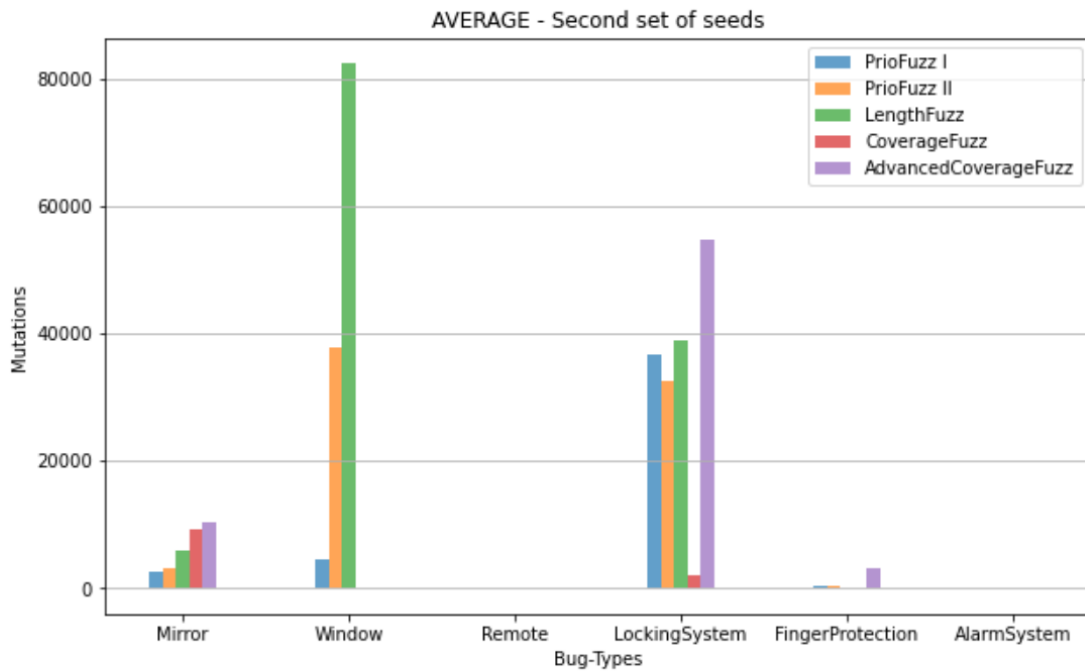

Figure 5.8: Second set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

5.3.3 Third Set of Seeds

Here we present the results of fuzzing with the third set of seeds. With eight seeds, the third set of seeds is much bigger than the previous two. Examining Figure 5.9, one notices how the Window- and LockingSystem-Bug are again very difficult to discover for all fuzzers. LengthFuzz and AdvancedCoverageFuzz could not find the Window-Bug at all in less than 100000 rounds of fuzzing when executed 200 times. All approaches besides AdvancedCoverageFuzz could find the other bugs during almost every execution. Very salient is AdvancedCoverageFuzz's ability to discover the AlarmSystem-Bug. While the other fuzzers found the AlarmSystem-Bug in all 200 rounds, AdvancedCoverageFuzz had only 66% probability of finding the bug.

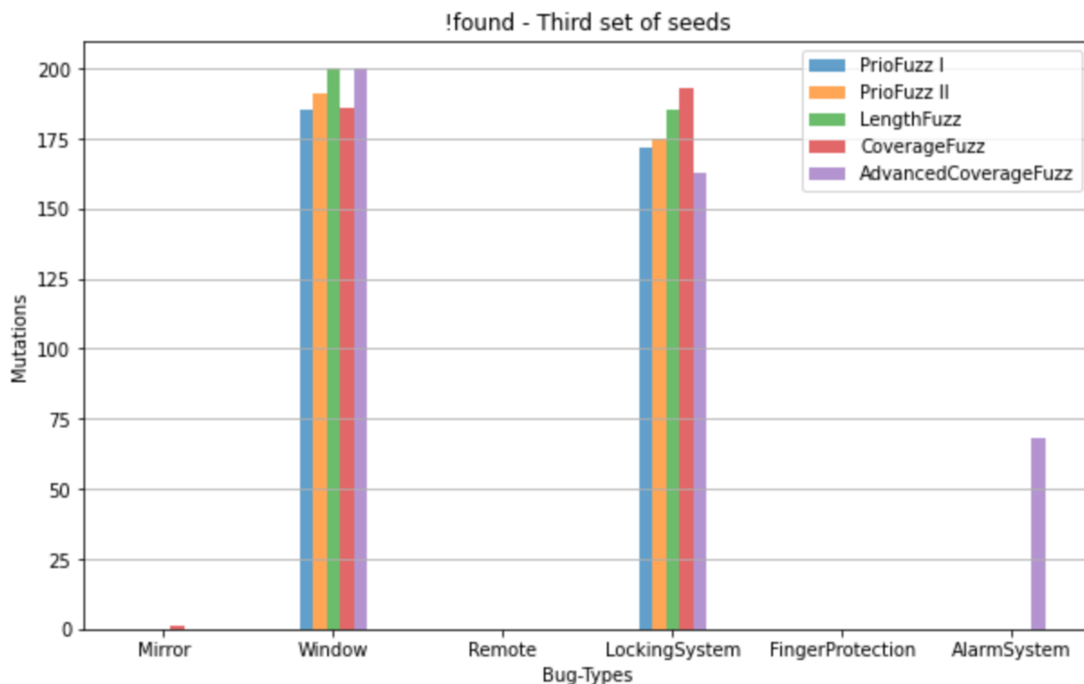

Figure 5.9: Third set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

Two details are salient in Figure 5.10, the diagram of minimum mutation amounts: PrioFuzz II was much worse at the Window-Bug and Length- and CoverageFuzz were much worse at the LockingSystem-Bug than the other fuzzers that were able to discover these bugs.

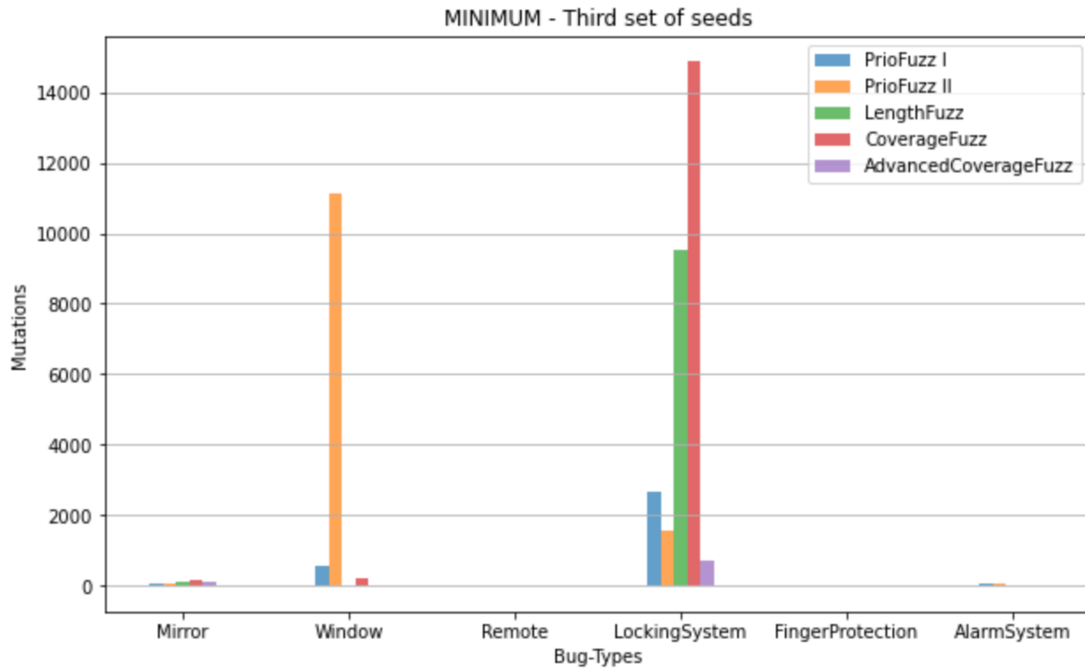

Figure 5.10: Third set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

More expressive statistics for the fuzzers performance are not the minimum amounts to find a bug but the measures of central tendency, which can be seen in Figure 5.11 and 5.12. The outlier of PrioFuzz II in the minimum amounts to find the Window-Bug is not supported by these diagrams. Remarkable is how CoverageFuzz performed more than thrice as good than both PrioFuzzers in median and more than twice as good in the arithmetic mean for the Window-Bug. On the contrary it performed about twice worse than the PrioFuzzers in finding the Mirror- and LockingSystem-Bug. While AdvancedCoverageFuzz was equally good as the PrioFuzzers, it performed more than ten times worse than LengthFuzz in finding the AlarmSystem-Bug.

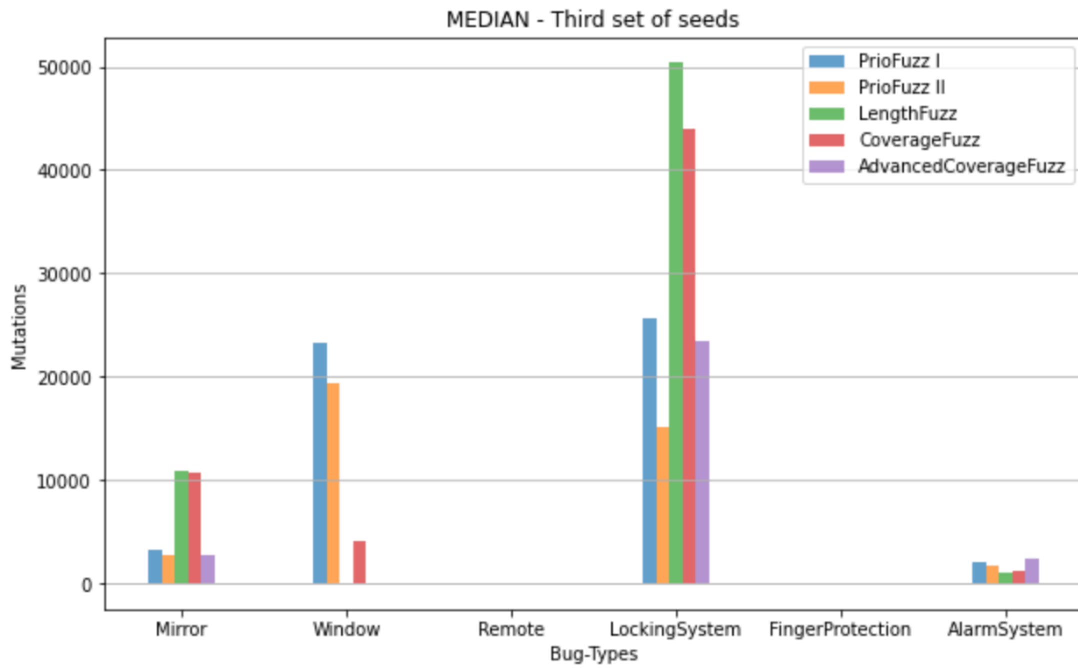

Figure 5.11: Third set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

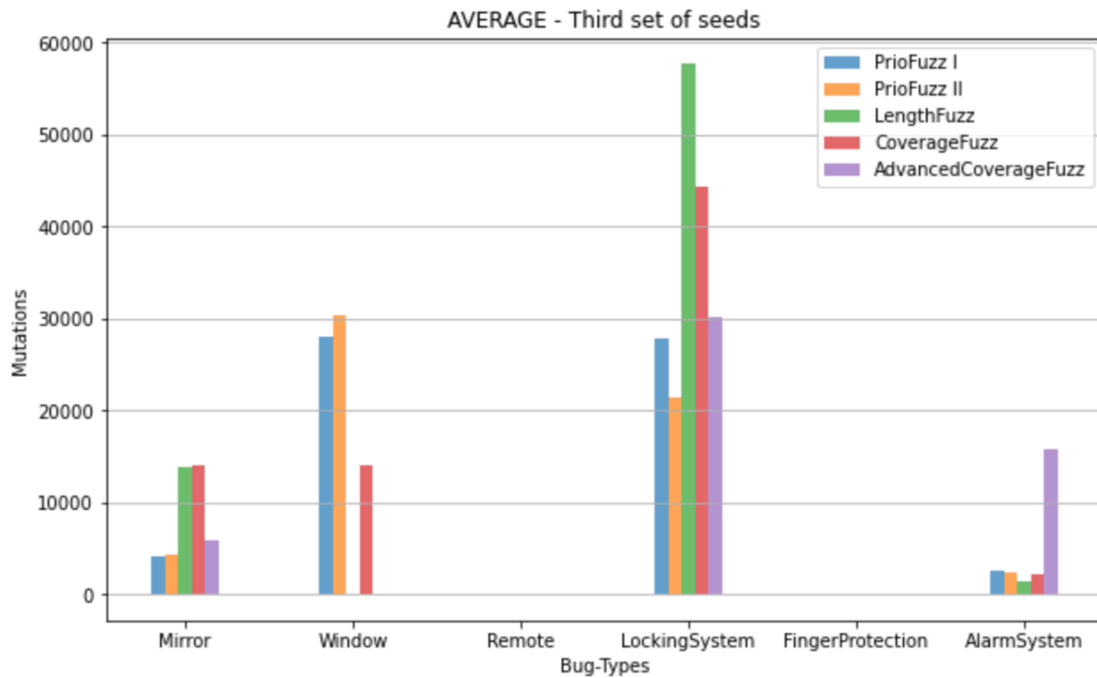

Figure 5.12: Third set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

5.4 Effect of Other Factors than a and b

After comparing PrioFuzz I and PrioFuzz II to the other three fuzzers, we ask ourselves if other properties of our approach affect the performance. We briefly look at the importance of parameter c, the values of categoryPrios, the removal of similar seeds and the reset of seed probabilities.

5.4.1 Removal of similar Seeds and Role of Parameter c

The Figure 5.13 shows how often the bugs were found. We compare two different c values to the value of 0.3 which PrioFuzz I and II use. This diagram also includes our approach without the removal of similar seeds. There are no significant differences between the smaller and greater c. The seed removal ensures that the set of seeds is not growing limitlessly. However, we see in Figure 5.13 that it is statistically of minor importance for the finding of these bugs. Figure 5.14 confirms the similarity of approaches with varying c. Also, it clearly shows that the seed removal does improve the median count of mutations needed in order to find any of the bugs. We defined a specific method for seed comparison which removes seeds that do not play a significant role for increasing the variety of the set of seeds and Figure 5.14 shows that the idea helped increase efficiency by two to three times in this case.

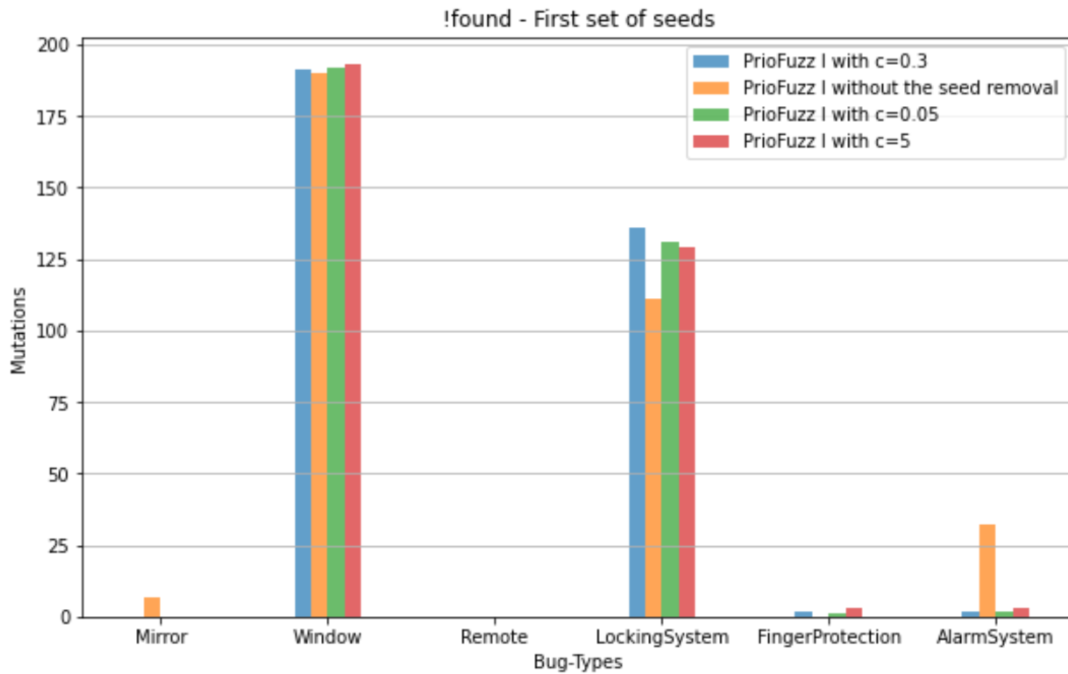

Figure 5.13: Role of c: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

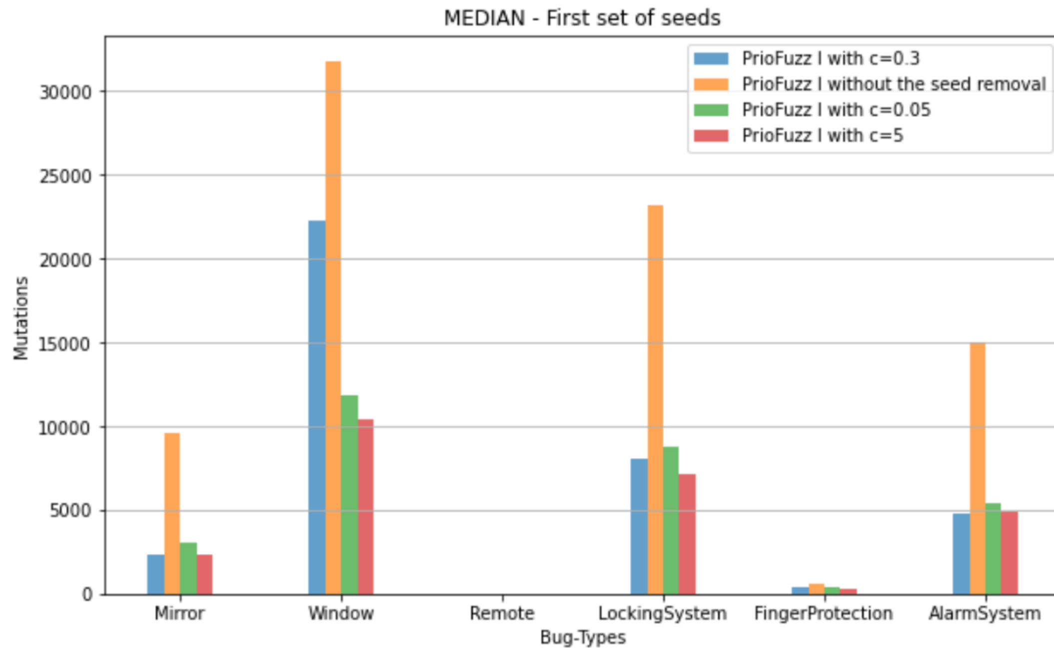

Figure 5.14: Role of c: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

5.4.2 Role of Category Priorities

We examined three different combinations of priorities for the categories HI, DS, SEC and SAF. Figure 5.15 shows that the amount of findings does not change significantly with different sets of values. However, Figure 5.16 illustrates the median of the different fuzzers and shows that there are bugs in which the category priorities achieve results two times better or worse than others. But there is no clear correlation between the features where the bug is located and the category priorities. This would require further studies to make conclusions.

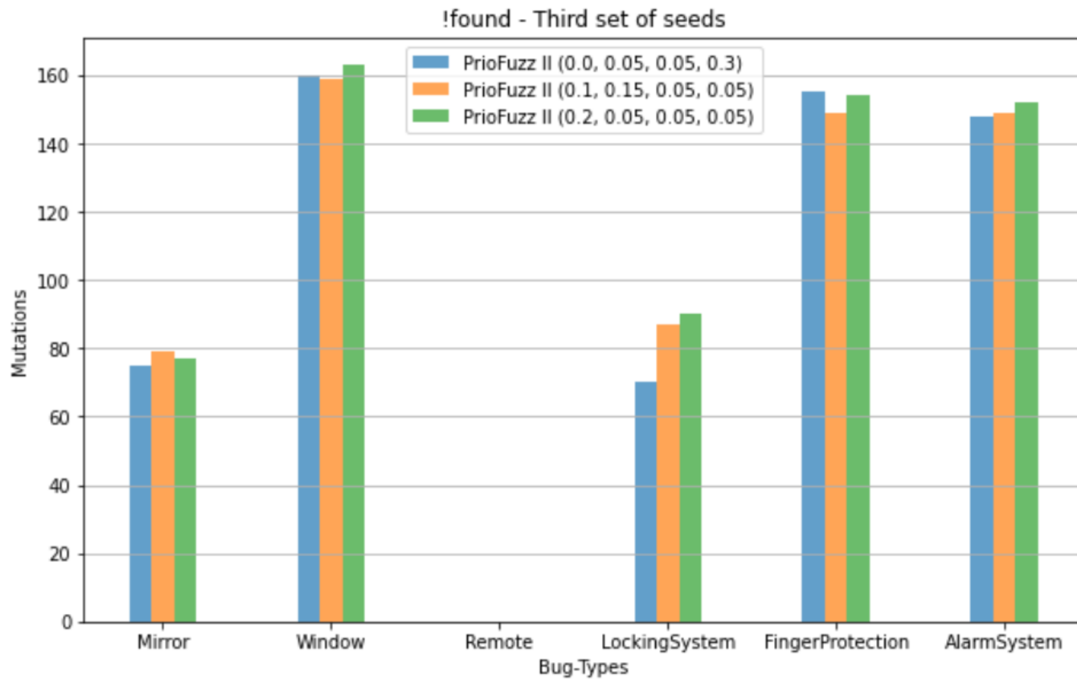

Figure 5.15: CategoryPrio: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

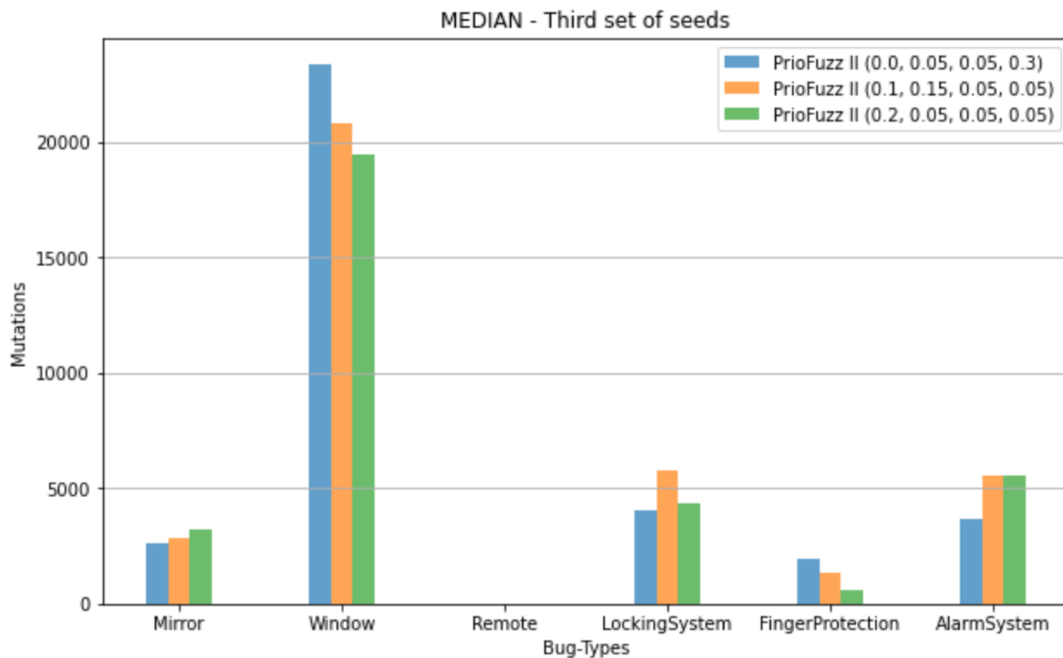

Figure 5.16: CategoryPrio: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

5.4.3 Repeatedly resetting the Probabilities in the Set of Seeds

We chose to reset the probabilities in the set of seeds if the fuzzer does no progress for too long. A reset means, all priorities are assigned the probability $\frac{1}{|\text{seeds}|}$. We are evaluating the influence of using this property, by comparing PrioFuzz II with it against PrioFuzz II in which we deleted the property for this subsection (PrioFuzz II ProbReset in Figure 5.17).

Figure 5.17 shows that without the occasional reset of probability, the fuzzer is less often able to find the bugs. The reason might be that the fuzzer is stuck with highly prioritized seeds which explore completely different parts of the SUT. In this respect, the reset increases effectiveness. However, when we look at Figure 5.18, we see that the approach which does not reset the probabilities at any time, achieves two times better median mutation counts than the normal PrioFuzz II.

Both variants might have their own advantages and disadvantages but there might be a correlation to the size of the set of seeds that has to be further examined. For the main part of evaluation, we decided to always use a reset which triggers after the fuzzer has not added any new seeds for too long.

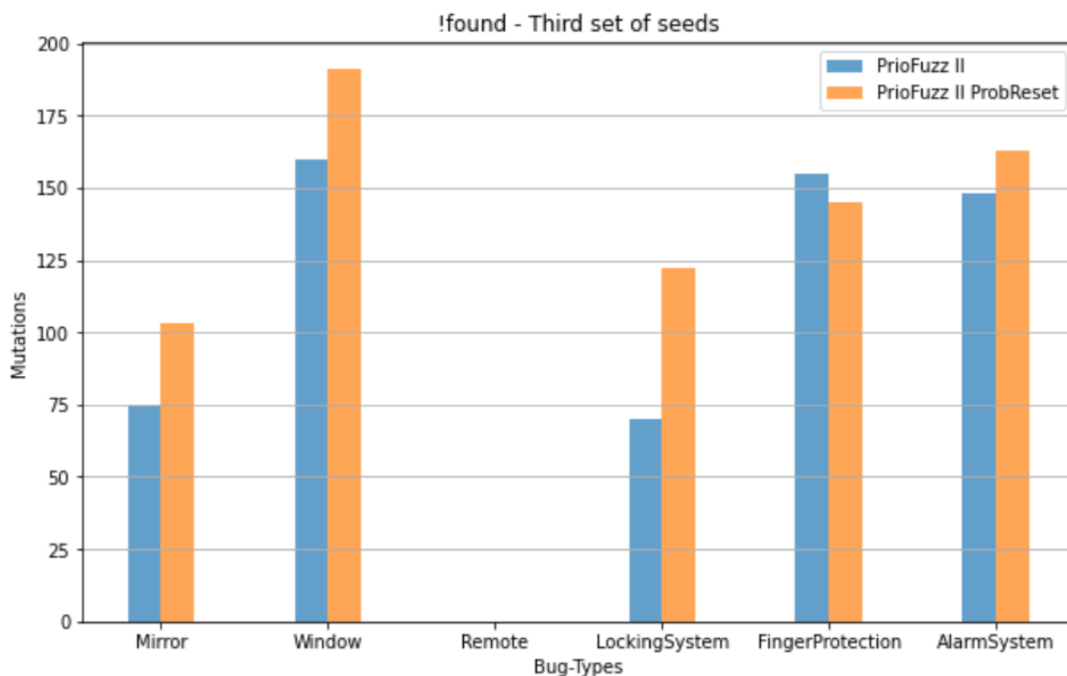

Figure 5.17: ProbReset: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

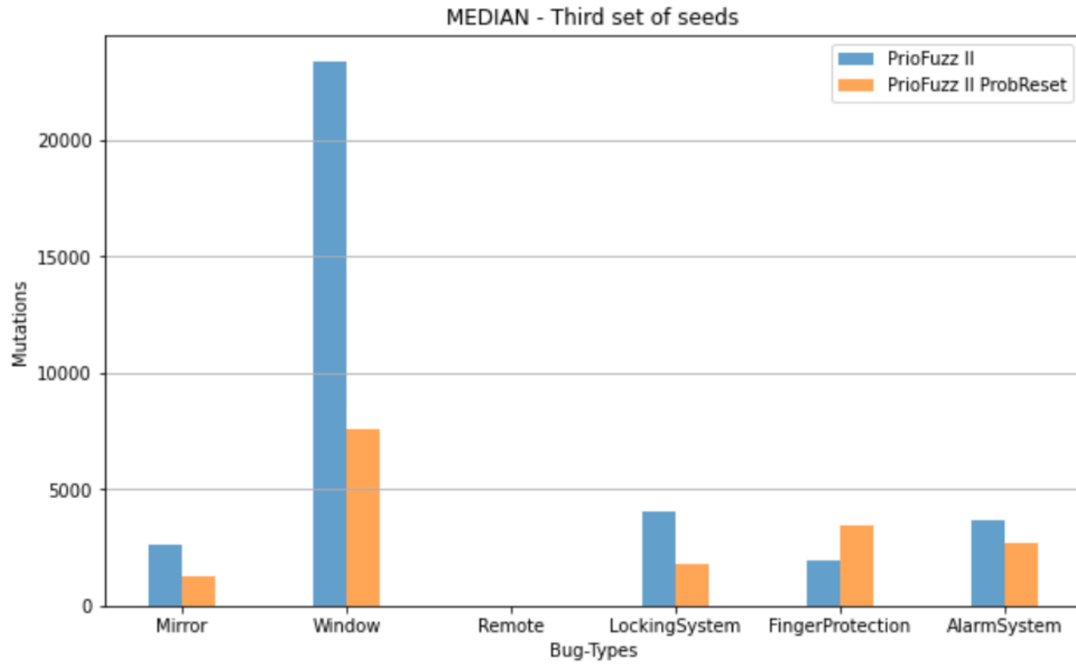

Figure 5.18: ProbReset: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

Summary

In this chapter, we implemented two versions of our proposed fuzzer and compared the results to three pre-existing approaches. The SUT is an instantiation of the BCS [LLLS17]. We compare several criteria:

- the amount of times out of 200 executions in which the fuzzers were not able to find a bug.
- the minimum amount of mutations needed to find a bug out of 200 executions.
- two measures of central tendency: the median and arithmetic mean.

Three different versions of the initial set of seeds are compared individually to exclude that the election of the initial set of seeds plays a more critical role for the fuzzing performance than the election of the fuzzing approach. In this chapter, we focussed on fuzzing the SUT under a constant order of input stimuli during every execution. However, we added the evaluation of fuzzing with randomized stimuli order in the Appendix A.1. We found that our optimized approaches achieved the best results overall under all comparison criteria with very few singular exceptions.

In the next Chapter, an extended interpretation and conclusion of our findings follows.

6 Conclusion

In this work, we introduced a novel fuzzing approach which allows the product-specific prioritization and categorization of features derived from variability models. The underlying approach is inspired by FuzzFactory [PLS+19], SlowFuzz [PZKJ17] and PerfFuzz [LPSS18]. Execution path length-based fuzzing resonates with our idea of feature priority and feature category: We count the execution path lengths within categories and increment the counters with priority-dependent values when basic blocks are visited. With the main idea remaining unchanged, mutated seeds are saved when they increase the execution path length of at least one category.

However, we determine the election probability of a seed based on the amount of increase it achieves. As a result, our approach not only favors seeds that extend the execution path length through exploration of many basic blocks, but also values seeds achieving considerable increases by visiting basic blocks associated with highly prioritized features. The following formula shows how the probability of a seed is calculated:

$$P(\text{seed}) = \max\{a + \text{categoryPrio} + b \times (1 - 2^{-c \times \text{categoryIncrease}}) \mid \text{all categories}\}$$

Chapter 5 compares some possible variants of parameter combinations. While minor differences were oberservable, both variants produced excellent results in the aggregate. However, within the scope of this work, a clear and general conclusion regarding which parameters produce the best results cannot be made.

Regarding our first research question, we presented commonly used goals for fuzzing, like coverage-guidance and execution path length, which were eventually used for the performance comparison.

After introducing different variability models, we presented the CVM inspired by Wittler et al. [WKR22] in order to ensure a stable foundation for the derivation of product-specific goals from variability models and therewith answering RQ2 with our optimization process in Chapter 4.

The third research question revolved around available fuzzing tools. We introduced a few examplary tools but due to the intricacy of embedded systems, we could not find an universally applicable fuzzer. For this reason, finding an appropriate fuzzer for SPL testing remains essential. We chose to develop our own tool in Java for the evaluation part of this thesis.

The comparison of our approach to a simple execution path length-based approach and coverage-based approaches showed, that our approach produced superior results in many situations but not in all aspects. Occasionally, the coverage-guided approaches were able to yield better results and sometimes even the unaltered execution path length-based approach would yield the best results.

When creating a fuzzing tool from scratch for testing a SPL, we are definitely recommending the application of our process despite the increased time cost. This is due to the required adaptations to a basic execution path length-based approach being minimal and the studying and augmentation of the variability models helping in getting a better understanding of the SPL. When the objective is

to test only a few instantiations of a SPL in a very short-termed testing project while there is an available and suitable fuzzer, it is probably less expensive and equally effective to run the unaltered fuzzer more often with the saved amount of development time. Ultimately, any fuzzer must be executed numerous times due to the pseudo-randomness affecting the set of seeds, the mutations and eventually the test results. For an ongoing long-term project of an SPL, we recommend an optimized fuzzer. Our proposal proved to be slightly more performant and could improve testing efficiency in the long-term.

Outlook

In this work, we proposed a process that assigns priorities to categories and features and allows the categorization of features. Our contribution involves a novel formula for the calculation of seed priorities and this proposal can be generally applied to SPLs.

The category- and increase-dependent assignment of priorities was enough for the fuzzer to outperform the simpler approaches but the exact calculation of probability values is yet to be optimized. Our findings suspect that the parameters do not affect the results very much but the formula allows various possible parameter combinations which could not be compared in detail in the scope of this work. It is to be examined if there might be an optimal parameter combination or if the parameters could be derived in a meaningful way from additional properties of the SPL. Also, the binary base in the exponential function is not the only possible base, a different base would satisfy the requirements of convergence as well.

Another possible adaptation of our process could be the change of instrumentation from execution path length to code coverage. With the idea of feature priorities it was intuitional to use execution path length for guidance. It would not be trivial but it could also be possible to redefine the formula and make new seed probabilities dependent on the transitions between features.

Bibliography

[Aus23] D. Aust. *Thesis: Fuzzing project*. Privately Published. Access by contacting st167292@stud.uni-stuttgart.de. 2023 (cit. on pp. 51, 55).

[Bet22] M. Betka. *Fuzzing I*. ASTA exercise 5, Institute of Software Engineering, 14.06.2022. 2022 (cit. on pp. 15, 25).

[Cod23] *Code Intelligence*. Accessed August 15th. 2023. URL: https://www.code-intelligence.com/ (cit. on p. 31).

[CWBE16] D. D. Chen, M. Woo, D. Brumley, M. Egele. "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware". In: *NDSS, Vol. 16*. 2016, pp. 1–16 (cit. on p. 30).

[Dmn21] *DMN – Decision Model and Notation*. Accessed August 7th. 2021. URL: https://www.bpmn.de/lexikon/dmn/ (cit. on p. 38).

[DMRJ13] D. Davidson, B. Moench, T. Ristenpart, S. Jha. "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution". In: *Presented as Part of the 22nd USENIX Security Symposium (USENIX Security'13)*. 2013, pp. 463–478 (cit. on p. 30).

[Eis22] M. Eisele. "Debugger-driven Embedded Fuzzing". In: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2022. DOI: 10.1109/ICST53961.2022.00062 (cit. on p. 32).

[EMS+22] M. Eisele, M. Maugeri, R. Shriwas, C. Huth, G. Bella. "Embedded Fuzzing: A Review of Challenges, Tools, and Solutions". In: *Cybersecurity* 5.1 (Sept. 2022), 18 pages. DOI: 10.1186/s42400-022-00123-y (cit. on p. 32).

[Gri17] D. Gries. *Testing Code Coverage in Eclipse*. Accessed August 7th. 2017. URL: https://www.cs.cornell.edu/courses/JavaAndDS/files/codeCoverage.pdf (cit. on p. 25).

[Ins23] C. M. U. S. E. Institute. *Software Product Lines Collection*. Accessed August 15th. 2023. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513819 (cit. on p. 15).

[KRC+18] G. Klees, A. Ruef, B. Cooper, S. Wei, M. Hicks. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. ISBN: 9781450356930. DOI: 10.1145/3243734.3243804 (cit. on p. 24).

[Kuh10] A. Kuhn. "On Recommending Meaningful Names in Source and UML". In: *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. RSSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 50–51. ISBN: 9781605589749. DOI: 10.1145/1808920.1808932 (cit. on p. 34).

[Lib23] *libFuzzer – a library for coverage-guided fuzz testing*. Accessed August 7th. 2023. URL: https://llvm.org/docs/LibFuzzer.html (cit. on p. 24).

[LLLS17] S. Lity, R. Lachmann, M. Lochau, I. Schaefer. “Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study”. In: *Technical Report 2012-07, Technische Universität Braunschweig, 2012*. 2017, 337 pages (cit. on pp. 17, 33–35, 41–45, 51, 68).

[LPSS18] C. Lemieux, R. Padhye, K. Sen, D. Song. “PerfFuzz: Automatically Generating Pathological Inputs”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 254–265. ISBN: 9781450356992. DOI: 10.1145/3213846.3213874 (cit. on pp. 27–29, 31, 47, 69).

[LZJ+19] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, J. Sun. “Polar: Function Code Aware Fuzz Testing of ICS Protocol”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.5s (Oct. 2019), 22 pages. DOI: 10.1145/3358227 (cit. on pp. 19, 31).

[Mar06] S. Marr. *Feature-Diagramme und Variabilität*. Accessed August 7th. 2006. URL: https://stefan-marr.de/pages/feature-diagramme-und-variabilitat/ (cit. on pp. 33, 34).

[Mül+09] T. C. Müller et al. *A Comprehensive Description of a Model-based, Continuous Development Process for AUTOSAR Systems with Integrated Quality Assurance*. Informatik-Bericht 2009-06. TU Braunschweig, 2009 (cit. on p. 34).

[Ost+11] S. Oster et al. “Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations”. In: *Proceedings of the 15th International Software Product Line Conference*. Vol. 2. SPLC ’11. ACM. Munich, Germany, 2011, 6:1–6:8. DOI: 10.1145/2019136.2019143 (cit. on p. 34).

[PBL05] K. Pohl, G. Böckle, F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005 (cit. on pp. 36, 38).

[PLS+19] R. Padhye, C. Lemieux, K. Sen, L. Simon, H. Vijayakumar. “FuzzFactory: Domain-Specific Fuzzing with Waypoints”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360600 (cit. on pp. 22–31, 49, 69).

[PSP18] H. Peng, Y. Shoshitaishvili, M. Payer. “T-Fuzz: Fuzzing by Program Transformation”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056 (cit. on p. 31).

[PZKJ17] T. Petsios, J. Zhao, A. D. Keromytis, S. Jana. “SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2155–2168. ISBN: 9781450349468. DOI: 10.1145/3133956.3134073 (cit. on pp. 27, 31, 47, 53, 69).

[Roo22] F. Roos-Frantz. “A Preliminary Comparison of Formal Properties on Orthogonal Variability Model and Feature Models”. In: *Universidade Regional do Noroeste do Estado do Rio Grande do Sul (UNIJU’I)* (2022). URL: http://www.gca.unijui.edu.br/members/frfrantz/publications/vamos-2009.pdf (cit. on p. 38).

[Sql23] *How SQLite Is Tested*. Accessed August 7th. 2023. URL: https://www.sqlite.org/testing.html#sql_fuzz_using_the_american_fuzzy_lop_fuzzer (cit. on p. 25).

[TDM08] A. Takanen, J. Demott, C. Miller. “Fuzzing for Software Security Testing and Quality Assurance”. In: (Jan. 2008) (cit. on p. 19).

[TK09] S. Theodoridis, K. Koutroumbas. *Clustering: Basic Concepts*. Accessed August 10th. 2009. URL: https://www.sciencedirect.com/topics/computer-science/hamming-distance (cit. on p. 48).

[Wei21] D. Weiskopf. *Graphen: Überblick*. Datenstrukturen und Algorithmen Vorlesung, 28.06.2021. 2021 (cit. on p. 22).

[Wil18] T. Wilson. “Evaluation of Fuzzing as a Test Method for an Embedded System”. In: (Apr. 2018), 43 pages (cit. on p. 20).

[WKR22] J. W. Wittler, T. Kühn, R. Reussner. “Towards an Integrated Approach for Managing the Variability and Evolution of both Software and Hardware Components”. In: *26th ACM International Systems and Software Product Line Conference - Volume B (SPLC ’22)*. ACM. Graz, Austria, Sept. 2022, p. 5. DOI: 10.1145/3503229.3547059 (cit. on pp. 37, 39, 69).

[Yak23] K. Yakdan. *What Bugs Can You Find With Fuzzing?* Accessed August 15th. 2023. URL: https://www.code-intelligence.com/blog/what-bugs-can-you-find-with-fuzzing (cit. on p. 15).

[YRKS22] J. Yun, F. Rustamov, J. Kim, Y. Shin. “Fuzzing of Embedded Systems: A Survey”. In: 55.7 (2022). ISSN: 0360-0300. DOI: 10.1145/3538644 (cit. on pp. 30–32).

[Zal14] M. Zalewski. *Pulling JPEGs out of thin air*. Accessed August 7th. 2014. URL: https://lcamtuf.blogspot.com/ (cit. on pp. 24, 25).

[Zal19] M. Zalewski. *AFL Documentation Release 2.53b*. Accessed August 17th. 2019. URL: https://github.com/google/AFL/releases (cit. on pp. 20, 23, 25, 31, 53).

[ZWG+21] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, M. Kim. “BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction”. In: ASE ’20. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 722–733. ISBN: 9781450367684. DOI: 10.1145/3324884.3416641 (cit. on pp. 15, 20, 30, 31).

All links were last followed on August 27, 2023, unless otherwise stated.

A Appendix

A.1 Fuzzing Results under random Input Orders

Contrary to the constant sequence from Section 5.3, a random sequence might never be able to reach certain code parts. Logically, all fuzzers should have worse finding counts for bugs that are at these certain code parts. We want to evaluate and compare the performances of the same fuzzers as in Section 5.3: PrioFuzz I, PrioFuzz II, LengthFuzz, CoverageFuzz and AdvancedCoverageFuzz. The difference is that a random sequence is generated before every execution. With random sequences, we expect the PrioFuzzers to perform equally well compared to the other fuzzers.

We utilize the exact same environment and testing conditions as earlier, with the only difference being the randomized sequences.

A.1.1 First Set of Seeds

As expected, Figure A.1 shows how all fuzzers find less bugs when not being fuzzed under a constant sequence. The results for the constant sequence are depicted in Figure 5.1. The Remote-Bug is not sequence-dependent and can still be found in every execution of fuzzing very easily. The Mirror-, FingerProtection- and AlarmSystem-Bug were found less often. The LockingSystem-Bug was found more often by all fuzzers. A reason could be that there are fewer input combinations triggering the bug in the constant sequence than on average with random sequences. Relatively compared to each other, all fuzzers remain the same in a ranking how often they find the bugs. The comparably improved ability by the PrioFuzzers to find the Window-Bug is counterbalanced by their degraded ability to find the AlarmSystem-Bug.

In order to make a point while comparing the diagrams of minimum mutations needed, we would have to increase the execution count. With only 200 executions, there are differences between Figure A.2 and Figure 5.2 which might disappear when evaluating the results of more executions. The diagrams are in most parts similar. The Window-Bug and AlarmSystem-Bug features unlikely differences.

There might be random sequences under which it is either easier or more difficult to trigger some of the bugs. However, we expect comparable results on average. In the following we compare the median values and mean arithmetic values in Figure A.3 and A.4 to Figure 5.3 and 5.4.

The median features only two salient details: the LockingSystem-Bug, where all fuzzers are slightly better under random sequences than under the constant sequence. As mentioned earlier, this bug was also found more often under random sequences. And the other detail is that the differences between the fuzzers in finding the FingerProtection-Bug are far less present under random sequences.

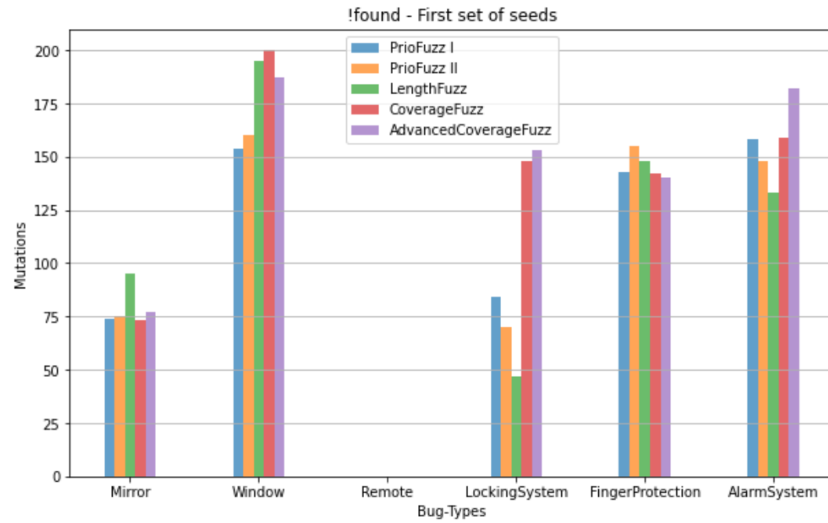

Figure A.1: First set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

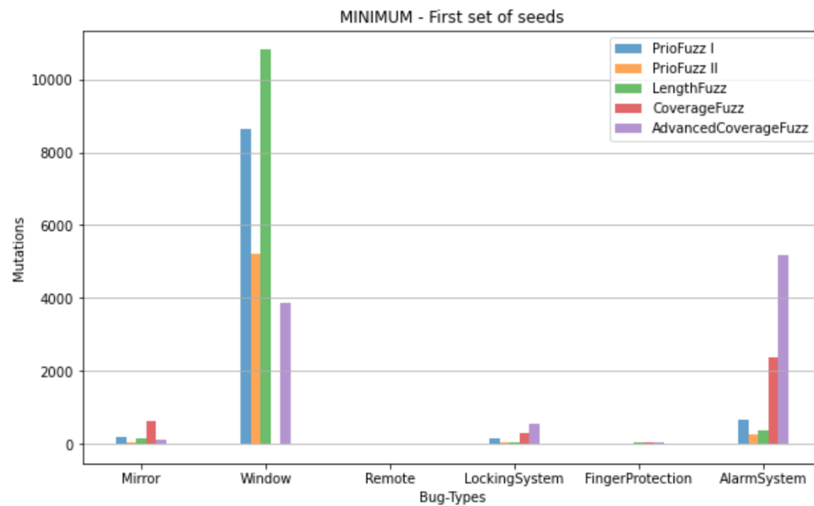

Figure A.2: First set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

The same applies to the arithmetic mean. The difference between these two measures of tendency is that the arithmetic mean is greater than the median for all bugs. The reason might be that there are a few outliers with very high mutations counts. The arithmetic mean is more sensitive to outliers than the median.

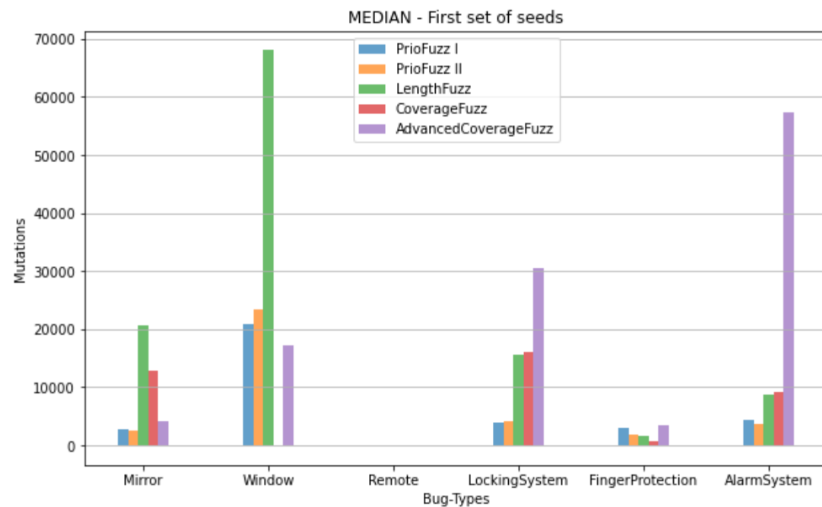

Figure A.3: First set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

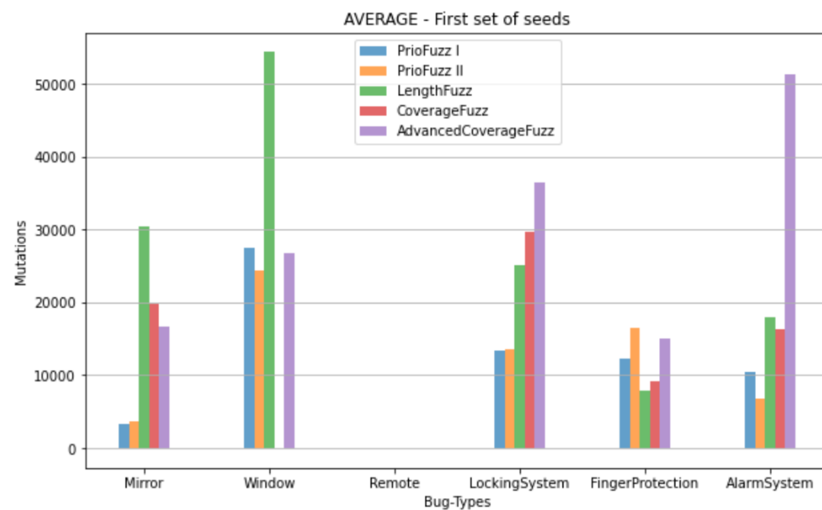

Figure A.4: First set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

A.1.2 Second Set of Seeds

With the second set of seeds, which includes three seeds and a few extreme values, there were overall better results in finding the Mirror-Bug and AlarmSystem-Bug. Figure A.5 also shows that the PrioFuzzers again achieved similar results in finding the Window-Bug. According to the data of Figure A.1 and A.5 that would mean that the PrioFuzzers find the Window-Bug more easily when fuzzing random sequence instead of one constant sequence as in Section 5.3. The difference to the constant sequence of the second set of seeds is enormous, since the two fuzzers found the Window-Bug just one time each. CoverageFuzz was the only fuzzer that improved in finding the LockingSystem-Bug. Here, the other fuzzers are slightly worse compared to the first set of seeds.

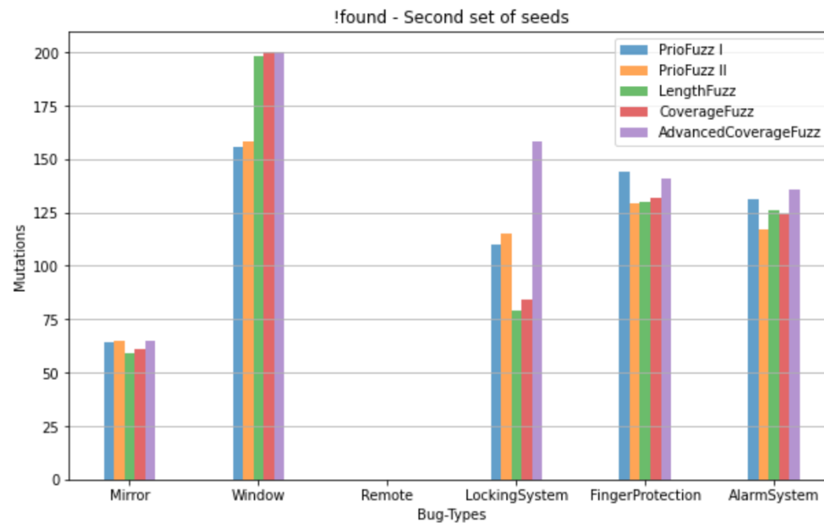

Figure A.5: Second set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

It is very eye-catching in Figure A.6 that the fuzzers found all bugs besides the Window- and LockingSystem-Bug in far less than 100 mutations at least once. These results are much better than the results of random sequences with the first set of seeds. However, with the second set of seeds, the minimum mutation counts were only slightly worse with the constant sequence.

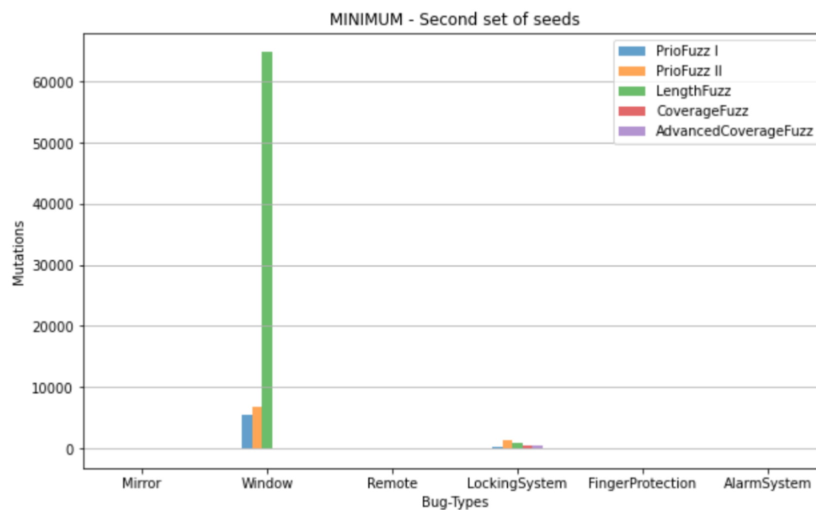

Figure A.6: Second set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

When comparing the medians of the constant sequence from Figure 5.7 to Figure A.7, the diagrams are mostly similar. One difference is that the PrioFuzzers performed much better under random sequences for the LockingSystem. The medians for the Window-Bug are different but that is due to the fact that the bug was only found once under constant sequence.

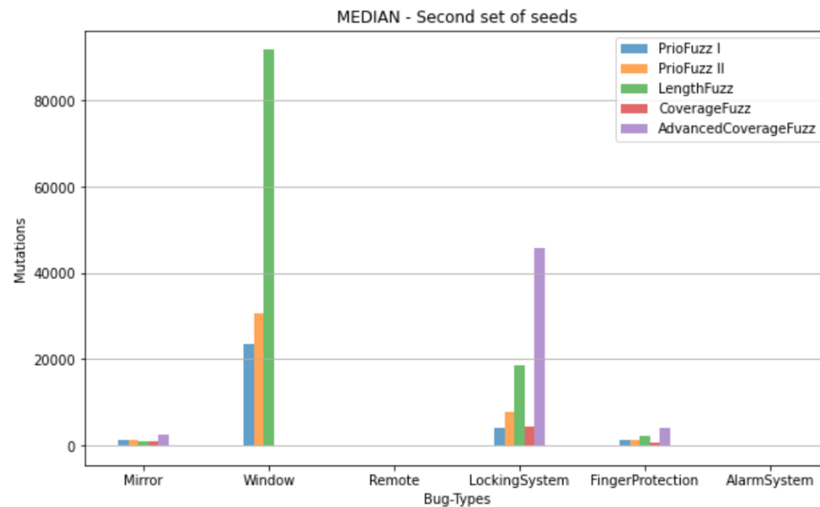

Figure A.7: Second set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

Looking at the arithemtic mean in Figure A.8, we see similar properties as with the median for the LockingSystem-Bug. While it was not evident that the fuzzers achieve worse results at the FingerProtection- and AlarmSystem-Bug with random sequences than with the constant sequence. The arithmetic mean highlights how the random sequences are not beneficial in this occasion.

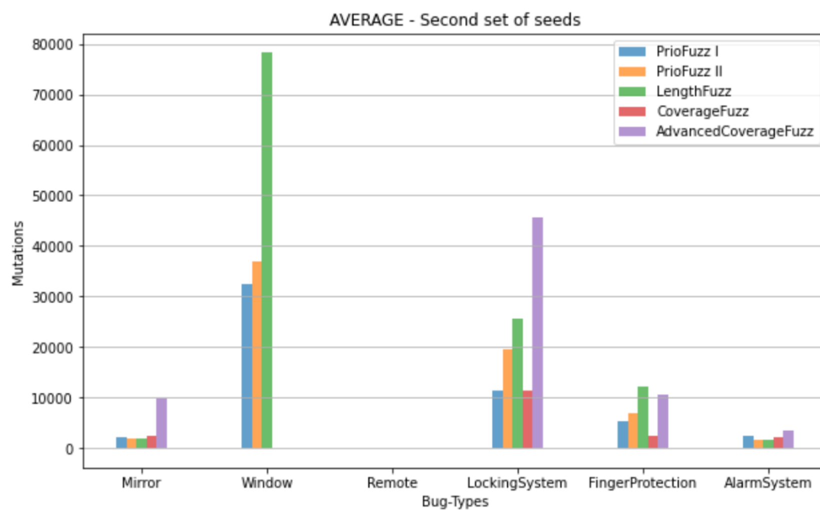

Figure A.8: Second set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

A.1.3 Third Set of Seeds

The third set is bigger than the first two. The previous observations hold when looking at the Figures A.9 - A.12. One new observation is that the coverage-based fuzzers were able to find the Window-Bug much more often than under any other circumstances. In this case, they achieved the best results with random sequences and the biggest set of seeds for the Window-Bug.

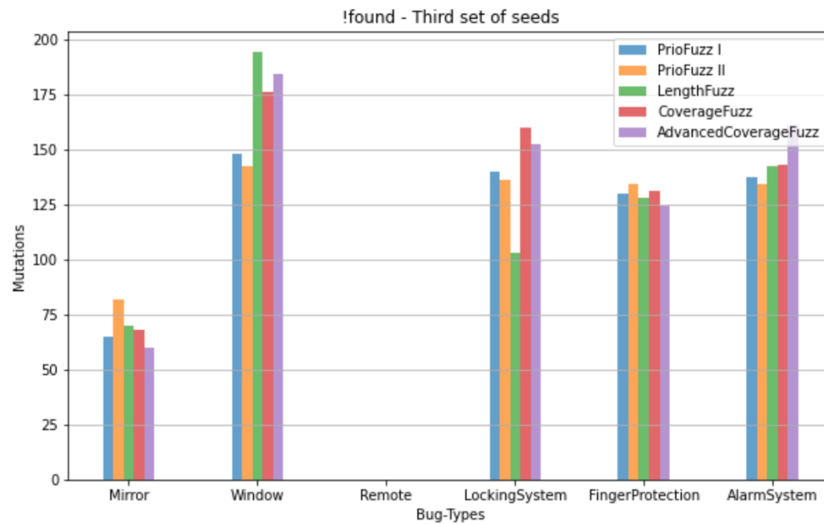

Figure A.9: Third set: The amount of times in which the bugs were not found. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

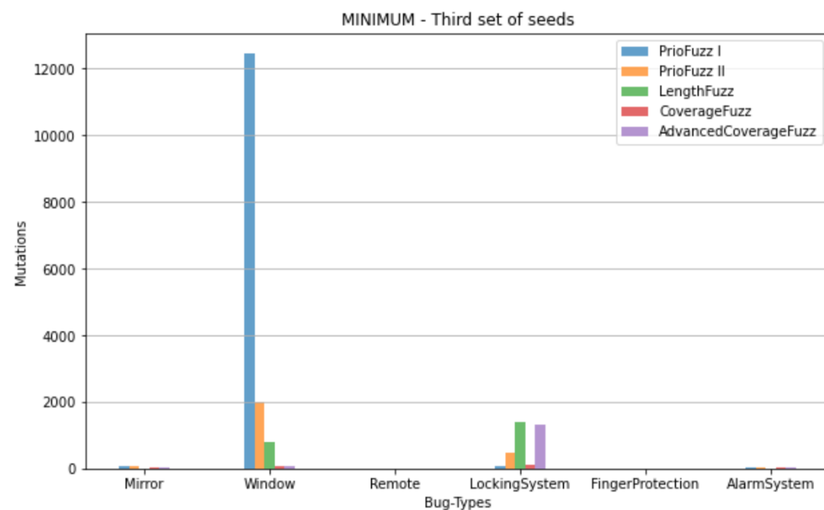

Figure A.10: Third set: The minimum amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

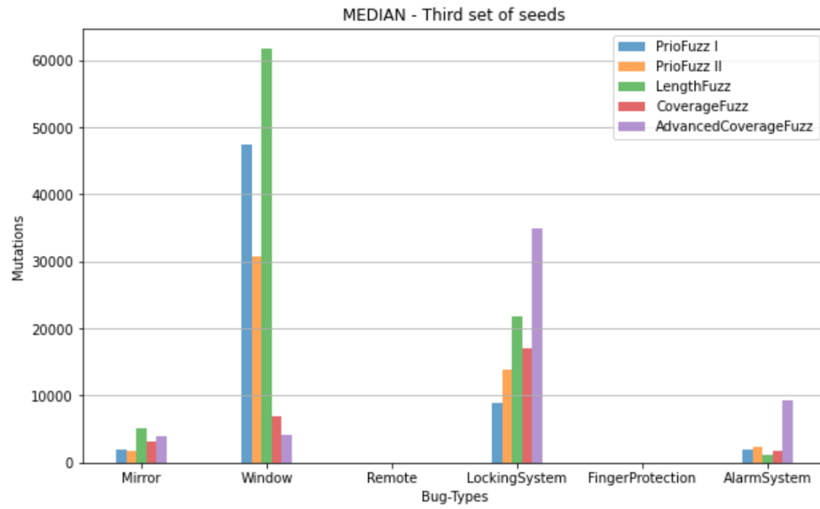

Figure A.11: Third set: The median amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.

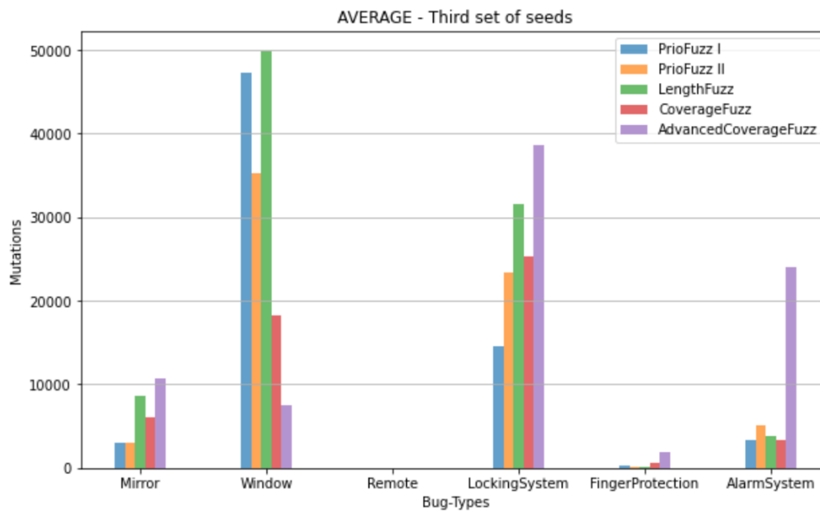

Figure A.12: Third set: The arithmetic mean amount of mutations needed to find the bugs. All fuzzers were executed 200 times. Mutation count threshold is 100000 rounds.