Introducing probabilities within grey-box fuzzing

Hänsynstagande till sannolikheter inom grey-box fuzzing

Patrik Sletmo

Supervisor: Ulf Kargén
 Examiner: Nahid Shahmehri
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Patrik Sletmo
Abstract

Over the recent years, the software industry has faced a steady increase in the number of exposed and exploited software vulnerabilities. With more software and devices being connected to the internet every day, the need for proactive security measures has never been more important. One promising new technology for making software more secure is fuzz testing. This automated testing technique is based around generating a large number of test cases with the intention of revealing dangerous bugs and vulnerabilities. In this thesis work, a new direction within grey-box fuzz testing is evaluated against previous work. The presented approach uses sampled probability data in order to guide the fuzz testing towards program states that are expected to be easy to reach and beneficial for the discovery of software vulnerabilities. Evaluation of the design shows that the suggested approach provides no obvious advantage over existing solutions, but also indicates that the performance advantage could be dependent on the structure of the system under test. However, analysis of the design itself highlights several design decisions that could benefit from more extensive research. While the design proposed in this thesis work is insufficient for replacing current state of the art fuzz testing software, it provides a solid foundation for future research within the field. With the many insights gained from the design and implementation work, this thesis work aims to both inspire others and showcase the challenges of creating a probability-based approach to grey-box fuzz testing.
I would like to thank all of the wonderful people who have helped me make this thesis work a reality. From Sectra Imaging IT Solutions AB I would like to give a special thanks to Andreas Ehrlund and Henric Granberg for their constant curiosity, kind support and great ideas. On the academic side I would like to thank my supervisor Ulf Kargén and examiner Nahid Shahmehri at Linköping University, your assistance and competence in the field has been invaluable for the success of this thesis work. Last but not least, I would also like to thank my family, friends and colleagues for their encouragement and support, for which I am forever grateful.
Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vi
List of Tables vii

1 Introduction 1
1.1 Motivation 3
1.2 Aim 4
1.3 Research questions 4
1.4 Delimitations 4
1.5 Method 5

2 Theory 6
2.1 Grey-box fuzz testing 7
2.2 Other fuzz testing approaches 13
2.3 Program analysis 13

3 Design and Implementation 19
3.1 Design 19
3.2 Implementation 27

4 Evaluation 32
4.1 Performance numbers 33
4.2 Power scheduling deviations 33
4.3 Results 34

5 Discussion 44
5.1 Results 44
5.2 Design 46
5.3 Implementation 50
5.4 Source criticism 52
5.5 Method 52
5.6 The work in a wider context 53

6 Conclusion 54

Bibliography 56
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Toy example used to illustrate the difference between black-box and grey-box fusers.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Psuedo-code for the evolutionary fuzzing architecture.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Simple Control-Flow Graph that has been partly explored by a fuzzer.</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Illustration of over- and under-approximation.</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Illustration of a program call graph and control flow graph. Rectangles represent functions and circles represent basic blocks. Call graph relations are shown using dashed arrows, and control flow graph relations are shown using filled arrows.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of a traditional fuzzing architecture and the architecture in Spare-Fuzz. Additions are highlighted in green and modifications are highlighted in yellow.</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Psuedo-code for the seed prioritization and corresponding score calculations.</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustration of power scheduling in SpareFuzz</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Coverage data for CGC_Hangman_Game.</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Coverage data for pre-seeded CGC_Hangman_Game.</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Coverage data for Character_Statistics.</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Coverage data for Multipass.</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>Coverage data for Multipass2.</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Coverage data for Palindrome.</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Coverage data for root64_and_parcour.</td>
<td>38</td>
</tr>
<tr>
<td>4.8</td>
<td>Coverage data for Sample_Shipgame.</td>
<td>39</td>
</tr>
<tr>
<td>4.9</td>
<td>Coverage data for Tennis_Ball_Motion_Calculator.</td>
<td>39</td>
</tr>
<tr>
<td>4.10</td>
<td>Area under curve distribution for CGC_Hangman_Game.</td>
<td>39</td>
</tr>
<tr>
<td>4.11</td>
<td>Area under curve distribution for pre-seeded CGC_Hangman_Game.</td>
<td>39</td>
</tr>
<tr>
<td>4.12</td>
<td>Area under curve distribution for Character_Statistics.</td>
<td>40</td>
</tr>
<tr>
<td>4.13</td>
<td>Area under curve distribution for Multipass.</td>
<td>40</td>
</tr>
<tr>
<td>4.14</td>
<td>Area under curve distribution for Multipass2.</td>
<td>40</td>
</tr>
<tr>
<td>4.15</td>
<td>Area under curve distribution for Palindrome.</td>
<td>40</td>
</tr>
<tr>
<td>4.16</td>
<td>Area under curve distribution for root64_and_parcour.</td>
<td>41</td>
</tr>
<tr>
<td>4.17</td>
<td>Area under curve distribution for Sample_Shipgame.</td>
<td>41</td>
</tr>
<tr>
<td>4.18</td>
<td>Area under curve distribution for Tennis_Ball_Motion_Calculator.</td>
<td>41</td>
</tr>
<tr>
<td>4.19</td>
<td>Deviation data for CGC_Hangman_Game.</td>
<td>41</td>
</tr>
<tr>
<td>4.20</td>
<td>Deviation data for pre-seeded CGC_Hangman_Game.</td>
<td>41</td>
</tr>
<tr>
<td>4.21</td>
<td>Deviation data for Character_Statistics.</td>
<td>42</td>
</tr>
<tr>
<td>4.22</td>
<td>Deviation data for Multipass.</td>
<td>42</td>
</tr>
<tr>
<td>4.23</td>
<td>Deviation data for Multipass2.</td>
<td>42</td>
</tr>
<tr>
<td>4.24</td>
<td>Deviation data for Palindrome.</td>
<td>42</td>
</tr>
<tr>
<td>4.25</td>
<td>Deviation data for root64_and_parcour.</td>
<td>42</td>
</tr>
<tr>
<td>4.26</td>
<td>Deviation data for Sample_Shipgame.</td>
<td>42</td>
</tr>
<tr>
<td>4.27</td>
<td>Deviation data for Tennis_Ball_Motion_Calculator.</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Measured execution time for the presented approaches ......................... 29
4.1 Time to discovery for the bugs found using the new heuristics (hh:mm) ........ 35
4.2 Time to discovery for the bugs found using the old heuristics (hh:mm) ........ 35
4.3 Peak coverage achieved using the new heuristics .................................. 40
4.4 Peak coverage achieved using the old heuristics .................................. 40
The use of fuzz testing, or fuzzing, for finding vulnerabilities is a growing trend in the software industry and an active area of research [1]. Both Google and Microsoft do, for example, utilize long-running fuzzing in order to detect vulnerabilities before malicious parties do. These fuzzing setups are running at a large scale in order to generate enough data to test as many program paths as possible. Microsoft published an article in 2012 on the challenges of running such a large fuzzing setup where they describe the infrastructure they use [2]. At that time, Microsoft used an average of 200 machines a day for running the fuzzing. Google did at the same time use “several hundred virtual machines” in order to fuzz test their Chrome web browser. Google’s blog post claiming this does, however, indicate that they quadrupled their capacity shortly thereafter [3].

Fuzz testing is inspired by the desire for robustness in applications. If an application can accept any data without crashing or malfunctioning, then it is indeed robust. Fuzz testing is a testing approach which repeatedly executes an application with unexpected data with the aim of crashing the application. This data is generated by mutating input files using various heuristics. If the test application frequently crashes when exposed to using fuzz-testing it is not very robust. However, fuzz testing provides no guarantee that an application is indeed safe if no bugs are found within a fixed time frame. The use of fuzz testing simply has the chance of discovering bugs if there are any.

Within fuzz testing, there are several areas that strike different compromises in order to find as many bugs as possible. Grey-box fuzzing is a balanced approach positioned between the more extreme black-box and white-box fuzzing approaches. Black-box fuzzing is the most simple form of fuzzing and does not make use of any program knowledge except for if the generated input caused the program to crash or not [1]. This can be problematic as the generated input tends to reach only shallow parts of the program under test. However, the technique is still successful in many areas because it can generate test cases at a quick pace. Some variations of black-box fuzzing also utilize input grammar to generate only data that conforms to the expected format, which helps increase the number of test cases that are testing interesting parts of the tested program [4].

---

1A program path represents the ordered set of code segments that are executed during a single run of a program.
A white-box fuzzer is, in contrast to black-box fuzzing, using a lot of knowledge about the tested program in order to make more informed decisions during fuzzing. For instance, the input data that makes the program execute a specific part of the program can be calculated during white-box fuzzing using symbolic execution and constraint solving [2]. While this is an effective way of reaching new program paths, it causes the test-case generation throughput to decrease significantly. This decrease of throughput can be some cases counteract the benefit gained from more informed decisions. Together with the complexity of having to model the entire program behavior, this causes white-box fuzzing to be impractical for use in the industry [1].

A grey-box fuzzer is instead more guided than the outcome-agnostic black-box fuzzer but has a higher test-case generation throughput than a white-box fuzzer. These properties come from the use of lightweight coverage instrumentation in grey-box fuzzing. By observing with what input the coverage of the system under test improves, a grey-box fuzzer can estimate what input is more likely to either produce even more coverage or find more bugs.

One of the more popular grey-box fuzzers is American Fuzzy Lop (AFL) [5]. AFL uses an evolutionary approach in which inputs that help discover more coverage are saved as seed inputs. These seeds are then chosen one by one for further mutations, creating a feedback loop in which interesting input helps generate more interesting input. The simple architecture in AFL consists primarily of three essential parts: seed prioritization, power scheduling, and mutation strategies. These architectural components have inspired research that, in different ways, attempt to improve them. Research on seed prioritization, or what input seed to choose from the set of all saved seeds, is concerned with ranking seeds based on how likely they are to lead to the discovery of additional interesting seeds. Power scheduling determines how many times a chosen seed is mutated before a new seed is selected, with the number of mutations commonly referred to as the seed energy. Mutation strategies concerns how exactly the chosen seed is mutated. AFL implements a lot of different heuristics for generating interesting mutations, while more advanced approaches take the program logic and data-flow into account [6, 7]. These advanced approaches enable precise mutations that are more likely to increase the coverage of the system under test.

While previous approaches to the problems of prioritizing seeds and assigning seed energy have been shown to work well, neither of them combines both knowledge on what interesting outcomes may appear when fuzzing a specific seed together with the probability of these outcomes appearing. AFLFast uses rudimentary probability modeling for determining what seed energy to assign but does not consider it for seed prioritization [8]. Sparks et al. instead propose a method for seed prioritization that favors mutation of rare but already explored paths by using sampled probability values [9]. CollAFL takes the potential outcomes into account when prioritizing seeds, but does so without considering that it may be challenging to achieve an outcome identified as desirable [10]. Zhao et al. recently modeled fuzzing as a random sampling process which enabled them to introduce the notion of probabilities for exploring unexplored branches [11]. However interesting, this data is not used to guide fuzzing itself, but instead to determine when fuzzing (i.e., random exploration) becomes unlikely to expose new program behavior so that a more advanced alternative can be used.

In general, many fuzzers consider low-frequency paths or hard-to-reach code especially interesting and therefore prioritize them during fuzzing [6, 9, 12, 13]. However, even if a path is non-frequent or hard to reach, there is no guarantee that fuzzing along it will be beneficial.
for the fuzzing. This lack of more informed decisions could potentially limit the effectiveness of grey-box fuzzers. While white-box fuzzing is criticized for the decrease in throughput caused by sophisticated methods, a more informed approach to grey-box fuzzing should be able to maintain a throughput similar to existing alternatives.

In this thesis work, a consideration of the probability of achieving new code coverage is introduced as a way to improve grey-box fuzzing. This introduction is accomplished by extending the idea from CollAFL [10] that every seed has a potential outcome with a consideration to probabilities, creating an estimation of the anticipated outcome for every seed. These anticipated outcomes are used to prioritize seeds that have a higher probability of leading to the exploration of new program paths. In addition, the power scheduling component is designed to estimate how many mutations are required for finding new coverage by using statistically expected values. These two approaches are compared to an alternative version that does not consider the importance of probabilities for either seed prioritization or power scheduling.

1.1 Motivation

Even if it is difficult to say what benefits fuzzing, the approach proposed in this thesis work is based on the intuition that more accurate metrics will improve the fuzzing. The goal of CollAFL [10] and many other fuzzers is to discover new and interesting seeds that cover new program paths. The scoring system in CollAFL favors seeds that have the potential of exploring a larger number of new program paths. This favoritism helps solve the lack of consideration of potential outcomes in previous works, but their approach is slightly optimistic. Since CollAFL does not consider the likelihood of exploring unexplored branches, every branch is treated as equally difficult to explore. This simplification has consequences on how efficiently CollAFL can explore a larger amount of the program space, since time may be wasted trying to explore branches with conditions that are difficult to guess. This consequence is not optimal when the goal is to discover new and interesting seeds that cover new program paths. By instead normalizing the scores assigned by the scoring system in CollAFL with how difficult the score is to achieve, the seed prioritization becomes more in line with this goal. This return-on-investment-like scoring should cause the fuzzer to prioritize seeds that have the potential to discover new and interesting seeds quickly.

Research on power scheduling often discusses the explore vs. exploit trade-off. This trade-off concerns the balance between exploring as much of the program as possible and trying to exploit (i.e., focusing on) what is already explored. A fuzzer typically assigns a smaller number of mutations to seeds during the initial explore phase, and transitions towards a more significant number of mutations during the exploit phase. AFLGo [14] implements this behaviour using a simulated annealing-based power schedule. The problem with the approach in AFLGo is the need for a pre-defined time-point at which the fuzzer transitions from exploration to exploitation. By introducing a concept of difficulties, or required effort, for exploring new code, the phase will change from exploration to exploitation as the difficulty increases. If the difficulties are empirically estimated when running the fuzzer, this approach becomes more dynamic than what is proposed in AFLGo.

1.1.1 Sectra Imaging IT Solutions AB

This thesis work is conducted at Sectra Imaging IT Solutions AB in Linköping, Sweden. Sectra manufactures medical equipment used to visualize radiology images and work with sensitive patient data. They have a great interest in improving their security assurance practices by utilizing fuzz testing but do not have the same capacity as shown beneficial by Google and Microsoft. By using a fuzz testing approach that more effectively uses the information at hand instead of relying on vast computational resources, Sectra could further improve the reliability of their products.
Many previous fuzzers are designed to run on Linux, but Sectra, among many other companies, is running their applications in a Microsoft Windows environment. This motivates the introduction of a fuzzing framework written for the Windows operating system that uses more accurate metrics to guide the fuzzing.

1.2 Aim

This thesis work aims to investigate if the consideration of probabilities during seed prioritization and power scheduling can improve the effectiveness of a grey-box fuzzer. This is achieved by introducing the fuzzer SpareFuzz and evaluating it against a version that does not use probabilities.

A secondary aim of this thesis work is to provide useful insights on what a typical grey-box fuzzer looks like and the challenges of implementing and evaluating one on Windows.

1.3 Research questions

The aim of this thesis work is achieved with the help of the following research questions.

1. How can the estimated probabilities of exploring unexplored branches be integrated into a grey-box fuzzer?

   While the probability estimation proposed by Zhao et al. [11] is straight-forward to implement on its own, the correct use of it within a grey-box fuzzing context is not as obvious. This research question aims to propose and motivate an appropriate use for this information when implemented in the seed prioritization and power scheduling components of a grey-box fuzzer.

2. How does the effectiveness of SpareFuzz compare to a fuzzer that does not estimate probabilities dynamically?

   By introducing constant probabilities for traversing every branch in SpareFuzz, the probability impact is essentially disabled. This modification makes SpareFuzz very similar to CollAFL that was introduced by Gan et al. [10], which allows for a fair comparison between the approaches.

3. How much energy is saved or wasted by using the power scheduling proposed in SpareFuzz instead of an exponential-increase strategy?

   Böhme et al. estimate the seed energy by exponentially increasing it every time the same seed is chosen [8]. This heuristic is claimed to be a fast way of estimating the expected value of the seed energy. SpareFuzz instead estimates this expected value using statistical data in order to increase the precision of the assigned energy (in relation to how much is required). This research question aims to answer if the method proposed in SpareFuzz is any better than what Böhme et al. propose by comparing them both to the number of mutations that were required in order to find new seeds during fuzzing. Preferably, the total amount of assigned energy should be as close to the required number of mutations as possible.

1.4 Delimitations

While important for the effectiveness of a fuzzer, this thesis work does not consider any of the other previously studied areas within grey-box fuzzing. This includes the areas of adaptive mutation and taint-based fuzzing. Adaptive mutation controls the granularity of mutations depending on the objective. Taint-based fuzzing can with high precision determine what part of the seed must be mutated in order to negate a branch condition.
In order to more easily evaluate the effectiveness of SpareFuzz, no other evaluation programs than a subset of the DARPA Cyber Grand Challenge (CGC) \cite{15} test suite will be used.

1.5 Method

The method used for conducting this thesis work consists of three parts: the design decisions, the implementation, and the evaluation of the fuzzer. The two larger parts, design and implementation, are detailed in the chapter “Design and Implementation” while the evaluation is further detailed in a separate chapter.

SpareFuzz is evaluated against a subset of the DARPA Cyber Grand Challenge (CGC) test suite. The CGC test suite has been used for evaluating fuzzers in the past, and SpareFuzz uses the same set of test programs as Li et al. used for evaluating their fuzzer Steelix \cite{16}.
Fuzz testing is in its most simple form a way to test how applications behave when exposed to unexpected, or fuzzy, data. The fuzz testing technique originates from an observation made during the late 20th century when a heavy storm caused data integrity errors in a remote terminal session [17]. The errors caused random characters to appear in the terminal prompt, which forced the user to either type their commands quick enough to avoid the extra characters or deal with the problems these extra characters caused. Surprisingly enough, these erroneous inputs did not only hinder the original intent behind the entered commands, but they also caused the underlying applications to crash. These unexpected reliability issues inspired Miller et al. to conduct experiments in which random data was intentionally given as input to various UNIX command-line applications. In total, over 24% of the tested applications crashed when exposed to the synthetic data, motivating further research on this new testing technique.

Since its first appearance, research on fuzz testing has given rise to many new variations of the crude method used by Miller et al. [1]. The technique has also received great interest from large software companies which use fuzz testing to test their applications for hidden issues continuously. What Miller et al. proposed is the purest form of black-box fuzzing, a fuzz testing technique similar to traditional black-box testing. Just like the testing approach, black-box fuzzing does not utilize any information on how the system under test (SUT) is implemented. This property allows the fuzz testing to be rather uncomplicated and at the same time proceed with high speed.

Despite its operational benefits, the simplicity of black-box fuzz testing makes it impractical for testing more complex applications. There are more advanced black-box fuzz testing approaches available, such as grammar and model-based fuzzing, but they are more tedious to configure [18, 4]. Consider a model describing the Portable Document Format (PDF), for example. The specification describing the format spans over a thousand pages [19], which makes formalizing the format in its entirety an enormous task. Even with such a model, black-box fuzzing fails to cover many corner-cases in complex applications because of its unguided nature.

The need for a more guided approach to fuzzing led to the emergence of grey- and white-box fuzz testing approaches, which both utilize information about the SUT to cover more program
2.1 Grey-box fuzz testing

Grey-box fuzz testing inherits many properties from black-box fuzz testing. When Sparks et al. presented the first variant of a grey-box fuzzer, they called it an extension of black-box fuzz testing, rather than a different approach [9]. However, a similar fuzzer presented in the same year was, perhaps more correctly, called a grey-box fuzzer [20]. The fuzzer presented by Sparks et al. is using an evolutionary algorithm to determine what fuzzed inputs are more likely to produce interesting results. By measuring basic coverage information, the fuzzer can be more efficiently guided than a black-box fuzzer, which only knows if the program has crashed or not. This coverage information can be obtained using lightweight instrumentation of the program binary that does not result in much overhead. The obtained information is used to sequentially guess the solution to individual constraints embedded in the program logic instead of having to guess multiple constraints at once. Consider the example in figure 2.1, in which a segmentation fault is triggered when the input string is equal to “bad”. A black-box fuzzer needs to guess the entire string at once and does not know how good or nearby a guess is. If the string “bac” is guessed by a black-box fuzzer there is no way for it to know how close it is. A grey-box fuzzer using an evolutionary algorithm would typically start by guessing random strings, just like a black-box fuzzer. When the first letter is correctly guessed to be “b” the first branch in the program is taken. This new branch coverage is detected by the fuzzer which saves the triggering input and uses it as a starting seed for new fuzzing rounds. By performing subsequent mutations on a string that has a correct first letter, it is more likely that the fuzzer continues to guess a correct second letter while keeping the first letter unchanged, compared to guessing these two letters correctly at the same time. The same principle is used to guess the last letter, which triggers the crash.

The advantages of the above toy example can be quantified by making a few simplifications. Suppose that both the black-box and grey-box fuzzer has an input space of three bytes. Then, an exhaustive enumeration of all the possible combinations by a black-box fuzzer would require $256^3 = 16,777,216$ guesses in the worst case. Suppose that the grey-box fuzzer instead iterates through the bytes in the seed and tries every possible combination for each byte. This iteration would instead require at most $256 \times 3 = 768$ guesses to reach the first branch. The input that reaches this branch would now be saved, giving a total of two seeds to fuzz. This would result in a maximum of $2 \times 256 \times 3 = 1,536$ guesses to reach the second branch. With three seeds to fuzz, the last branch would require at most $3 \times 256 \times 3 = 2,304$ guesses. In to-
2.1. Grey-box fuzz testing

Q = GetInitialSeeds()
repeat
    S = NextSeed(Q)
    E = CalculateEnergy(S)
    for i from 1 to E do
        M = Mutate(S)
        O = ExecuteProgram(M)
        if IsCrash(O) then
            Save M
        else if IsInteresting(O) then
            Add M to Q
        end
    end
until aborted by user

Figure 2.2: Pseudo-code for the evolutionary fuzzing architecture

tal, the grey-box fuzzer requires at most 4,608 guesses before the bug is found. This is several magnitudes faster than a non-guided black-box fuzzer.

In addition to evolutionary fuzzing, a technique called taint-based fuzzing is also included within the grey-box fuzzing field. This technique utilizes knowledge of what part of the input data ends up in a specific program location in order to fuzz with precision. The taint-based fuzzer BuzzFuzz identifies potential attack vectors in the program code and attempts to fuzz the input arguments to these functions [6]. VUzzer instead uses taint information to focus the fuzzing on input that is used to compute branch conditions, increasing the probability that a branch condition is modified as a result of fuzzing [12].

2.1.1 Evolutionary fuzzing architecture

While the architecture of a typical evolutionary fuzzer is simple, it is not quite as simple as the example in the above comparison. There are a few key components of an evolutionary fuzzer that appears in most implementations: a circular seed queue, a seed energy metric, a mutation algorithm, and a metric to determine if a seed is interesting or not [8]. The seed queue is initially seeded with some base seeds that aim to get an as diverse coverage of the SUT as possible. This seed queue is then looped through repeatedly, and every chosen seed is given an energy. This energy determines for how long the seed is mutated before a new seed is selected from the queue. For every mutation, the seed is mutated according to a mutation algorithm. If the mutated seed causes the program to crash, it is saved as output. If the mutated seed does not crash the program but is deemed interesting, it is added to the seed queue. In general, a seed is deemed interesting if it increases the coverage of the SUT. This loop continues until the user aborts it. An overview of the process can be seen in figure 2.2.

The prioritization of the seed queue and energy metric have received much attention in research on coverage-based and directed fuzzing [21]. Both of these fuzzing techniques are concerned with reaching specific program locations during testing. Coverage-based fuzzing aims to cover as many program locations as possible, while directed fuzzing aims to reach a limited set of program locations. Such a set could for example contain program locations that are likely to contain bugs.

Meanwhile, the mutation algorithm used in fuzzers has been researched more in the context of taint-based fuzzers [6] [12], but has also been considered important by Chen et al. in the context of directed fuzzing [21]. Whether a seed is interesting or not has not received as
2.1. Grey-box fuzz testing

much attention in research, but usually boils down to if the input is different enough from what is already in the current seed queue [8].

Seed prioritization

The seed prioritization in a grey-box fuzzer determines what seed to mutate next. Depending on the fuzzer design, this prioritization works in different ways. Böhme et al. provide an overview of how the fuzzer AFL and their derivative of it, named AFLFast, are implemented [8]. AFL implements a circular queue in which seeds are chosen in the same order as they are added. To optimize this somewhat, AFL classifies some seeds as favorites. When selecting the next seed from the queue, non-favorite seeds are often skipped. Chen et al. criticize this lack of proper prioritization in AFL because it creates a delay before newly found seeds are chosen [21]. AFLFast instead chooses the next seed based not only on the queue order and the favorite classification but also based on two metrics called $s(i)$ and $f(i)$.

- $s(i)$ represents the number of times the seed has been fuzzed.
- $f(i)$ represents the number of inputs that exercises the same path as the seed.

The following seed in the queue with the lowest $s(i)$ is chosen first, and to distinguish between several seeds with equal $s(i)$ the fuzzer chooses the seed with the lowest $f(i)$. The use of these metrics helps prioritize inputs that have been fuzzed fewer times and that exercise low-frequency paths, something that, according to Böhme et al., helps create a more effective fuzzer. Rawat et al. also identify that prioritizing low-frequency paths is beneficial for increasing the effectiveness of a fuzzer [12]. They note that error handling code is usually what is executed most frequently because the number of invalid inputs sent to the SUT far outnumbers the number of valid inputs. Since a valid input has a higher probability of triggering an interesting program behavior, it makes more sense to focus the resources on fuzzing such input. In their fuzzer VUzzer, they identify error handling code and actively discard inputs that reach these sections. This is in essence what is accomplished in AFLFast, only that AFLFast uses a heuristic for achieving similar behavior.

The focus on low-frequency paths has also influenced the fuzzer Angora by Chen and Chen [13]. By identifying all of the unexplored branches along the program path every seed it executes give rise to, Angora is able to choose branches to explore in its queue instead of seeds to mutate. Every seed that exercises a path adjacent to an unexplored branch is tied to this branch. Using this relation, Angora can mutate seeds that cause execution to pass by an unexplored branch until a new branch is chosen from the queue. By repeatedly choosing unexplored branches from the queue and attempting to explore them, the fuzzer gradually explores the program state space. Since the fuzzer consequently focuses on these unexplored branches, the authors claim that Angora focuses on low-frequency paths after exploring the high-frequency ones.

Hawkeye, which is a directed grey-box fuzzer, uses somewhat different means of prioritizing the seed queue [21]. The fuzzer prioritizes newly discovered seeds that either cover new program paths, are similar to a path used to reach the target location, or reach the target. Hawkeye only uses coarse comparisons for prioritization to avoid the algorithmic complexity of inserting an item in a priority queue. Seeds are added to queues of different tiers depending on their likelihood of producing new interesting seeds. Seeds that fulfill all of the criteria above are added to the first and most important tier. Seeds that are only newly discovered but do not meet any of the other criteria are added to the second tier. Other seeds are added to the third and final tier. Hawkeye will start by picking seeds from the most important tier, and only if that tier is empty, it will resort to the next.
2.1. Grey-box fuzz testing

Gan et al. utilize the precision improvements they introduced for coverage tracking to implement three new strategies for seed selection in CollAFL [10]. These three strategies were divided into two categories based on what the authors believed would improve using the proposed strategies. The first two strategies were designed to drive the fuzzer towards non-explored paths:

- **br** Prioritize seeds that contain many non-explored neighboring branches. Given a fixed theoretical chance of negating a conditional branch which the seed exercises, a larger number of branches would intuitively lead to a higher probability of negating any such branch.

- **desc** Prioritize seeds containing non-explored neighboring branches that may reveal many descendants paths. This strategy follows the same intuition as **br** but instead considers the number of descendant paths that can be exposed when exploring the neighbor branches.

A practical distinction between **br** and **desc** can be explained with the help of figure 2.3. Here, the circles marked with letters represent end-nodes of previously executed paths while the circles marked with numbers and their descendants represent unexplored parts of the program. Given the algorithm **br**, path A has a score of 1, path B has a score of 2, and path C has a score of 3. This score is obtained by counting the number of unexplored neighboring branches. Path A has the neighbor 1; path B has the neighbors 1 and 2; and path C has the neighbors 1, 3, and 4. If the **desc** algorithm is used instead, path A has a score of 1, path B has a score of 4, and path C has a score of 3. This is because node 1, 3, and 4 all reveal one new path, while node 2 reveals three new paths. Therefore, the algorithm **br** prioritizes path C, while **desc** prioritizes path B.

The authors also present a strategy with the goal of increasing the effectiveness of vulnerability discovery:

- **mem** Prioritize seeds that have a large number of memory access instructions. This follows the intuition that many vulnerabilities are triggered by insufficiently protected memory read or writes.
After a thorough comparison of the three strategies, the authors show that -br and -desc both find a similar number of vulnerabilities, while at the same time outperforming AFL and AFLFast. For improvement to coverage alone, -br outperforms -desc. The algorithm -mem performs better than the original algorithm implemented in AFL but is not as effective as the other two algorithms that the authors propose.

**Power scheduling**

The power scheduling controls how long a seed is mutated before a new seed is selected from the queue. The energy assigned by power scheduling is used to make sure that enough mutations are made for discovering new coverage. This energy is also used to limit how much time can be spent mutating a seed that has a low likelihood of actually generating interesting inputs. Böhme et al. identify that AFL has poor power scheduling that often assigns either too little or too much energy to a seed [8]. This flaw results in AFL both spending too much time on seeds that would produce interesting input with much less effort, as well as not spending enough time on certain seeds for interesting inputs to be discovered. In addition to this, the same seed is given similar energy every time it is selected from the queue. Böhme et al. claim that this is bad for two reasons: the energy is not increased for seeds that fail to generate interesting inputs, and it causes high-frequency paths to be exercised more times than needed. In their fuzzer AFLFast, they overcome these shortcomings by assigning energy proportional to the number of times a seed has been fuzzed, assigning less energy to high-frequency paths, and assigning more energy to low-frequency paths. The power schedule for computing the seed energy in AFLFast has been properly evaluated by the authors by comparing its performance with other power schedules. The authors show that the power schedule used has a good balance of speed and persistence, leading it to outperform every other power schedule evaluated in a 24 hour run with respect to the number of crashes caused.

In the comparison of power schedules for AFLFast there are two implementations that are typical within grey-box fuzzing: exploit and explore [8]. Exploit is what AFL uses by default, which, as noted above, assigns similar energy values to the same seed every time it is selected from the queue. Explore is similar to exploit but assigns less energy, causing the fuzzer to execute more seeds per time unit. The authors note the explore schedule results in the exposure of many crashes early on, but that the frequency of the exposures diminishes later on. AFLFast uses this property in its power schedule to assign exploration level energy for newly discovered seeds. When the seeds are later fuzzed further, the energy assigned increases in order to combat the diminishing returns of the ordinary explore power schedule. The increase in energy is exponential by design in order to quickly approximate the smallest energy needed to find new and interesting seeds. The decision to do so comes from the reasoning that the expected energy \( E[X] \) required to explore an unexplored branch is equal to \( E[X] = 1/p(t) \) where \( p(t) \) is the probability of traversing the branch. Since this probability is unknown in advance, AFLFast only uses it for modeling the problem. By doubling the assigned energy of a seed every time it is selected from the queue, the value \( E[X] \) can be quickly estimated without wasting too much energy.

A directed fuzzer derivative of AFLFast, named AFLGo, has a power schedule that is inspired by the properties of the exploit and explore power schedules [14]. AFLGo implements a simulated annealing-based power schedule that is dependent on the time spent fuzzing. When the fuzzing is started, all seeds are assigned approximately the same low energy, regardless of their properties. This behavior represents the exploration phase. As time passes, the seed properties start to make a difference. In its directed nature, AFLGo keeps track of how far a seed is from reaching the target location. Seeds that are far from the target are assigned lower energy as time passes, while seeds that are closer to the target are assigned higher energy. This strategy results in a higher probability to stress interesting behavior close to the target.
2.1. Grey-box fuzz testing

location, which is similar to the exploitation phase. The breaking point for when the fuzzer transitions from the exploration phase to the exploitation phase is configured beforehand.

Chen et al. suggests four ideal properties of a directed grey-box fuzzer and evaluates how well AFLGo matches these properties [21]. They claim that the power schedule used in AFLGo is an effective strategy for achieving the goal of reaching the target sites rapidly. However, the power schedule fails to avoid bias to individual traces. This bias can cause AFLGo to not put as much effort into mutating longer paths that still reach the target. Chen et al. emphasize the risk of this by showing that these paths may be the only paths that expose a vulnerability. They propose a solution to this that includes a covered function similarity metric within the power schedule. This metric is based on the intuition that seeds covering a similar set of functions as any of the possible paths that lead to the target have a larger probability of generating interesting mutations, and thus should be assigned more energy. Since this metric favors long paths, the authors claim that their addition mitigates most of the bias towards shorter ones.

2.1.2 Evaluation

The evaluation of fuzz testing frameworks in literature is largely sub-optimal. A recent literature study of 32 fuzzing papers conducted by Klees et al. shows that each of the studied papers had problems in their evaluation method [22]. One of the most important insights from their literature study is that some evaluation methods may not be as representative for estimating how good a fuzzer is. Many fuzzers are evaluated based on how much coverage they achieve, but while higher code coverage does correlate with more bugs being found, the correlation is moderate at best [23]. The authors behind AFLGo correctly identify that coverage (or reaching more targets in the same time) is mainly a measure of efficiency, while effectiveness is better measured using the number of vulnerabilities revealed [14]. Klees et al. claim that counting the number of found vulnerabilities is the best method for evaluating the performance of a fuzzer, but note that this is non-trivial. Ideally, the test suite would contain known bugs that can be easily distinguished when fuzzing.

LAVA-M is an artificially created test suite that contains several injected bugs, each with a unique identifier that is presented upon exploitation. The test suite is originally a collection of four GNU Coreutils but modified using LAVA (Large-scale Automated Vulnerability Addition) to contain easily distinguishable bugs [22]. Because the test suite originates from a set of programs written for UNIX, the LAVA-M test suite does not compile on Windows.

Another artificial test suite that behaves similar to LAVA-M is the programs used in the DARPA Cyber Grand Challenge (CGC). Instead of originating from actual real-world programs, the CGC test suite has been artificially created for the challenge. As an additional element in the challenge, the programs in the CGC test suite are written for the DECREE operating system. This design decision is motivated by two main reasons: it made existing tools challenging to use during the competition and made the operating system easy to model because of the small number of system calls available. The simple nature of the DECREE operating system has led to the successful porting of the CGC test suite to Linux, macOS and Windows [24].

The problem with both of these test suites is that the bugs are artificially created. Therefore, they are not very representative of real-world applications. However, because of the difficulty involved in deduplicating crashing inputs in real-world applications, Klees et al. still recommend the use of these test suites.

In addition to the advice on evaluation metrics and test suites, Klees et al. recommend that evaluation of a fuzzer contains the following elements:
2.2 Other fuzz testing approaches

There are some examples of different approaches to evolutionary fuzzing, which are more heavily inspired by the natural evolution process. Instead of prioritizing seeds and assigning energy to them, a fitness score is calculated for every seed. Seeds with high enough fitness are selected for breeding with other seeds. This breeding consist of a cross-over of two seeds together with minor mutations to the resulting seed. Sparks et al. present a directed fuzzer based on these principles that model the SUT as a dynamic Markov process [9]. The fuzzer models the process by tracking the coverage and updating the probabilities in the Markov chain accordingly. The authors introduce a fitness heuristic that is based on the inverse probability of reaching a given seed. This heuristic makes the fuzzer prioritize unexplored and hard-to-reach paths.

Another work that utilizes probabilities in the fuzzing logic is the hybrid fuzzer DigFuzz by Zhao et al. [11]. Instead of calculating a fitness based on previously recorded probabilities, DigFuzz estimates probabilities of unexplored paths to determine when and where to apply concolic execution instead of fuzzing. This estimation of probabilities is made possible by viewing the fuzzing process as a random sampling process. The recorded probabilities can then be used to estimate unexplored paths using the statistical rule of three [25]. The rule states that the probability \( p(e) \) of a binomially distributed event that has not occurred yet can be estimated by the number of times \( n \) that it has not occurred using the formula \( p(e) \leq \frac{3}{n} \). The formula gives an upper limit of the 95% confidence interval for the probability of the event. It should, however, be noted that the formula only has a statistical meaning once \( n \) is larger than 30. By utilizing this approximation, DigFuzz is able to precisely decide what unexplored paths may benefit from concolic execution instead of fuzzing alone. The authors claim that this is more efficient than running traditional hybrid fuzzing until the fuzzer gets stuck, and only then use concolic execution. The claim is motivated by a proper evaluation of how well the traditional method works in practice.

2.3 Program analysis

The principle of program analysis aims to give approximations of the behavior of a program by analyzing it either statically or dynamically. The results of such analysis can be utilized throughout the life-cycle of an application for tasks such as compiler optimization and validation [26]. Ideally, the program analysis would be exact (that is, both sound and complete). However, a precise analysis of the behavior of every program is not possible, as shown by Turing [27]. In his work, Turing formally proves that there can be no algorithm which answers whether a program terminates or not, which proves that a precise analysis is impossible. Following the discovery, the problem of determining if a program terminates has later been referred to as the halting problem [28].

---

1 Concolic execution uses a thorough program analysis in order to model and propose a solution to the conditions leading up to a particular program location. This makes it possible to reach a program location by solving an algebraic equation instead of relying on the randomness involved in fuzzing.
An analysis of the program behavior can be either sound or complete. A sound analysis is an over-approximation of the behavior, while a complete analysis is an under-approximation, as illustrated in figure 2.4. This distinction is important when reasoning about the potential impact a particular behavior has on properties such as program faults. A sound analysis might show faults in behavior that are not present in the real program. However, a complete analysis might instead fail to show actual faults that are indeed present in the real program.

2.3.1 Approaches

Within the field of program analysis, there are a couple of different approaches for analyzing a program. In order to facilitate different needs, tools have emerged which analyze programs in all of its forms: source code, intermediate representation, and machine code [29].

Source code

The analysis of source code may appear to be the most natural one since that is what programmers work with themselves. However, the complexity of today’s programming languages makes this analysis complicated. In an article on their commercialized bug-finding tool, Bessey et al. describe the various challenges of analyzing source code in a practical setting [30]. The main issue with analyzing code in a research setting is to understand the semantical meaning of the code, which is complicated in itself. The authors note that this is an issue in a practical setting as well, but that the code must be parsable before any semantical meaning can be deduced. The task of correctly parsing the code may seem simple, but because of low conformance to the C standard in compilers, there have emerged code that does not conform to the standard but still compiles. In an article published in the Microsoft C++ Blog on how to make Microsoft Visual C++ (MSVC) code compatible with the compiler Clang, the nonconformity to the standard in compilers is partly attributed to improvements made to the development workflow [31]. One such improvement is the possibility to deduce information instead of requiring the developer to specify it explicitly.

Intermediate representation

As described in The Architecture of Open Source Applications, many compilers utilize an architecture consisting of a front-end and back-end [32]. The front-end compiles the source code into an intermediate representation, and the back-end compiles this intermediate representation to machine code. This means that the front-end does not need to care about any architectural details, while the back-end can focus on only understanding the intermediate representation instead of all of the different programming languages which the compiler supports. An intermediate representation is in general on a lower level than the program-
2.3. Program analysis

This property makes intermediate representations especially suitable for program analysis because the analysis tool does not need to understand the semantics of a high-level programming language or the architectural details of machine code. Moreover, the intermediate representation is often using a reduced instruction set instead of a complex instruction set.

The LLVM intermediate representation (LLVM IR) is an intermediate representation that is commonly used for program analysis purposes [33]. LLVM IR has been constructed to assist the main objective of the LLVM compiler, which is to support life-long optimization of programs compiled with it. In order to realize this objective, LLVM IR embeds information to make program analysis easier. This information includes explicit control flow information and virtual function tables.

One example of a tool using LLVM IR for program analysis is Picon [34]. Picon is a tool presented by Coudray et al. that aims to provide Control Flow Integrity (CFI) by instrumenting the binary during compilation. The authors chose to use LLVM IR for this because of its simplicity compared to source code, its independence on both language and architecture, and structured, analysis friendly, form. Using the LLVM IR, Picon is able to perform the entire analysis statically and with high precision. The authors stress that any mistake during the analysis could lead to an incorrect termination of the instrumented program since a normal program path would be detected as illegal. This strict requirement puts some limitations on their tool, which means that not every program can be instrumented with it. Specifically, the authors have chosen not to instrument programs that make use of indirect jumps or calls. This category of calls is something that is typically used for virtual method resolution in C++ classes. The authors do, however, mention the existence of different ways to overcome this limitation.

Machine code

In order to fully capture the exact behavior of a program, the analysis can be performed on its resulting machine code. By doing so, all of the optimizations performed by the compiler and all of the architecture-specific logic that has been applied to the code is captured in the analysis. However, while capturing this information may be necessary to detect the most complex bugs, it comes at a cost. Machine code running on a personal computer (PC) is typically represented in either the x86 or the x86-64 instruction set, both of which are complex instruction sets. The x86-64 instruction set does, for instance, contain over 3000 instructions [35]. This great number of instructions would all need to be formalized in order to model the behavior of a program properly.

Even if the implementation of a tool designed to analyze machine code may be cumbersome, certain situations benefit from this analysis. Yun et al. presents several reasons to why concolic execution traditionally performs poorly in their paper introducing the hybrid fuzzer QSYM [36]. The authors argue that one of the most significant factors which make concolic execution slow is the emulation of an intermediate representation of the program that is required to model the behavior properly. This slowdown is attributed to the reduced instruction set used in intermediate representations, which increases the number of instructions in a program by an average factor of 4.69 times. By instead emulating the complex instruction set which the analyzed program was initially compiled for, the performance could be significantly improved. For a hybrid fuzzer such as QSYM, this improvement could lead to the discovery of software vulnerabilities in days instead of weeks.

\[^2\]Jumps where the target address is computed during runtime, rather than hardcoded in the binary.
2.3. Program analysis

Figure 2.5: Illustration of a program call graph and control flow graph. Rectangles represent functions and circles represent basic blocks. Call graph relations are shown using dashed arrows, and control flow graph relations are shown using filled arrows.

2.3.2 Call Graph

In program analysis, the relations between functions are commonly stored in a call graph, see Figure 2.5. Coudray et al. describes the call graph as a directed graph from caller basic blocks to called functions [34]. Because of difficulties in analyzing program behavior, the call graph usually does not contain any indirect jumps. Indirect jumps are not statically encoded into either the program binary or any intermediate representation, and can also be difficult to deduce even from the source code. Since indirect jumps are used for common language constructs, such as calling of virtual methods in C++, the lack of them makes the analysis unsound.

Babić et al. mitigate the inaccuracy caused by missing indirect jumps by dynamically collecting them as an initial preparation step [37]. By tracking the coverage when executing the SUT with a set of seed files, their approach can detect jumps that were not modeled in the original call graph. These detected indirect jumps are then added to the call graph before the testing continues.

2.3.3 Control Flow Graph

Just like the call graph models relations between functions, the control flow graph models relations within functions. Coudray et al. defines the control flow graph as a directed graph between different basic blocks within a function [34]. Every basic block except the entry block is preceded by at least one other basic block. If a basic block ends with a branch (i.e., a control-flow alternating statement), it is followed by two or more basic blocks. Basic blocks that end with a return statement does not have any successor, as the execution immediately continues in the caller function. An example control flow graph is illustrated in figure 2.5.

2.3.4 Program instrumentation

In order to record coverage during program execution, the program must be instrumented. There are multiple approaches to program instrumentation in literature. Sparks et al. track coverage by attaching a debugger to the SUT and set breakpoints at the entry-point of every interesting basic block [9]. This debugger has a custom breakpoint handler that keeps track of every visited node and uses this for construction of the execution path.
Coudray et al. instrument their programs by adding coverage tracking code during compile time [34]. This approach is made possible by the LLVM Compiler Infrastructure Project, which makes it possible to create custom plugins and optimization passes that run during compile time. These plugins and optimization passes have access to the intermediate representation used by LLVM and can analyze, modify, and add functionality to the compiled program. Their implementation is created as an LLVM plugin instead of forking the LLVM code base and adding it there, which has consequences on some of the available functionality. Even if both implementation alternatives use the same API in LLVM, there is a difference in the scope of the analysis. External plugins are only aware of the currently compiled file and not the program as a whole. Coudray et al. note that this limitation creates problems when instrumenting an entire program, and have gone great lengths in order to mitigate it. By instead forking the LLVM code base, the instrumentation code can be added as a Link-Time Optimization (LTO) pass. These passes consider the entire application as one module to allow for intermodular optimization, which completely mitigates the problems Coudray et al. experienced [38].

Since program instrumentation using LLVM is done during compile time, it creates complications for applications that are already compiled. In such cases, other tools such as Pin [39] or DynamoRIO [40] can be used [41]. These tools instrument binary applications by dynamically adding code to them. The fuzzer AFL uses QEMU user-mode emulation for instrumentation, which the author claim is faster than both Pin and DynamoRIO. Meanwhile, the Windows fork WinAFL uses DynamoRIO for instrumentation and quick restarts of the SUT [42].

A program can be instrumented with different levels of granularity during fuzzing. Gan et al. identifies the pros and cons with basic block coverage, branch coverage, and path coverage [10]. Basic block coverage is the most simple coverage metric, which simply records what basic blocks have been visited. This simple metric does, however, fail to capture many details. Since the same basic block can be visited in different ways, basic block coverage only captures what was visited and not how. By instead tracking the traversed branches this issue is mitigated, but information is still lost due to the lack of consideration to the order in which the branches are traversed. This can be solved by tracking the entire path that the program execution results in, but because this path can grow very long, it is not feasible to do so.

Chen and Chen also identify that the branch coverage is imprecise, but instead of proposing path coverage, they propose to include the execution context within the branch coverage information [13]. In AFL every branch is identified by a hash value that is computed using identifiers of the basic blocks that are immediately preceding and succeeding the branch. Chen and Chen complement this hash value by including the program stack leading up to the branch within the calculation. This way, calls to the same function from different locations are made distinct which allows for the separation of two otherwise seemingly identical seeds.

Another source of information loss is caused by hash collisions when calculating branch identifiers. The fuzzer AFL stores the hashes in a 64 KB bitmap for the sake of performance, as it allows for the bitmap to fit within the L2 cache [35]. Gan et al. identify that fitting within the L2 cache is indeed crucial for the performance of the fuzzing, but that the 64 KB bitmap causes collisions that limit the fuzzer [10]. They minimize the probability of collision by both modifying the hash computation algorithm and by adjusting the size of the bitmap when necessary. By introducing three new variables that are unique for every branch into the hash calculation, the potential collisions can be adjusted for and avoided. Since choosing these variables is a non-trivial task, they are only used whenever needed. The authors do, for example, note that many basic blocks are only preceded by one basic block, which makes it possible to calculate the hashes of these branches during static analysis. As these make up over 60% of all branches, this reduces the number of times that a solution to the three variable equation must be found. In some situations, these three variables can not be selected such that no collisions occur. In order to mitigate these situations, the authors create a hash table
with unique hashes that are indexed using the basic-block pair. During runtime, the hash is located within this table instead of calculated on the fly. The authors stress that this is much slower than calculating the hash because of the inevitable performance bottleneck caused by accessing memory.
This chapter describes the design and implementation details of SpareFuzz. First, the design decisions that make SpareFuzz different from previous works are outlined. Afterwards, the implementation details and the considerations made during the implementation follows.

3.1 Design

The design of SpareFuzz is split into different dimensions. First of all, the challenges that motivate SpareFuzz are described together with their proposed solution within SpareFuzz. This is intended to give a high-level understanding of the main ideas behind SpareFuzz. Afterwards, the general architecture is described and compared to the architecture of a typical evolutionary fuzzer. Lastly, the details of the different components used within SpareFuzz are described.

3.1.1 Motivation

SpareFuzz is created with the aim of improving the solutions to two challenges in a typical grey-box fuzzer: the problem of selecting which seed to fuzz next, and the problem of determining what number of mutations to assign to the chosen seed. Most solutions to these problems in previous works are based on simple heuristics based on basic interpretations of how programs behave. Intuitively, a solution to these problems based on an interpretation that closer resembles reality should perform better. SpareFuzz introduces the use of sampled data to calculate and estimate probabilities for traversing different program paths within the tested program. This introduction aims to reduce the gap between the interpretation and reality that is present in current solutions to the targeted problems.

Earlier work by Gan et al. [10] introduces new algorithms for seed prioritization that better approximate reality than previous works within this fuzzing area. These algorithms observe what program path every seed traverse, and uses knowledge of the program control-flow graph to determine how many unexplored program paths lay adjacent to them. Seeds with program paths having more unexplored paths adjacent to them are prioritized before seeds.

---

1 An adjacent program path is explored by flipping any of the conditions along the original program path.
with a fewer amount of adjacent paths. This method is more informed than relying on simple heuristics for finding new and interesting seeds, such as the prioritization of low-frequency paths that is present in AFLFast [8]. However, it is optimistic to believe that all adjacent paths have the same probability of being explored. SpareFuzz aims to improve the work by Gan et al. by extending it with probabilities that help approximate the effort (i.e., the number of mutations) required to explore unexplored program paths. SpareFuzz focuses on trying to explore paths that are easy to explore, with the intuition that it will lead to quicker discovery of even more paths, which in turn should benefit fuzzing. Since the probabilities are sampled continuously, paths that are found challenging to explore will be deprioritized until other, more easy-to-explore paths have been discovered. This dynamic behavior does not exist in the algorithms proposed by Gan et al., causing their fuzzer to spend much energy on the same set of seeds. Since some conditions in a tested program may be impossible to exercise using fuzzed input, which can cause the fuzzer to get stuck in a local maximum. Such never-met conditions could, for example, be tests to see if an input file failed to open, or if external configuration options are present.

Within the field of determining what number of mutations to assign a chosen seed, there is less data-driven research done. Böhme et al. [8] identify that the power scheduling, which assigns energy to seeds, in AFL is suboptimal. They claim that the power scheduling in AFL either assigns too little energy for new coverage to be discovered or too much, causing the fuzzer to waste time that could be better used elsewhere. Böhme et al. argue that an expected value of the required energy should be used instead. Their idea is good, and close to how informed an approach to this problem can be without resorting to more advanced program analysis techniques. The problem is that Böhme et al. approximate the expected value by exponentially increasing the assigned energy every time the same seed is chosen to be fuzzed again instead of calculating a more exact value using sampled information. Their approach is a suboptimal heuristic for approximating the value. SpareFuzz is instead using sampled probabilities to estimate the expected value. This is still a heuristic for approximating the required effort, but it is more informed than what is proposed by Böhme et al. while still based on the same theoretical reasoning.

As mentioned earlier, the probabilities of exploring different unexplored paths are not equal. This means that the problem of assigning seed energies must consider that seeds might have multiple unexplored branches, each with their distinct probability of exploration and corresponding effort estimation. Instead of merely choosing the minimal or maximal estimated effort of exploration based on values from all unexplored adjacent paths, SpareFuzz considers what effort level is anticipated to reveal as many new program paths as possible with as little effort as possible. This ensures that enough energy is assigned to reveal new program paths while at the same time avoiding the point of diminishing returns. Once the ratio between the potential outcome and the spent energy has reached its peak, it is no longer worth mutating the seed, and a new seed is chosen.

3.1.2 Architecture

The high-level architecture of SpareFuzz is designed to quite closely reassemble what a typical fuzzer looks like, but adapted for the individual needs of the fuzzer components described later. The previously presented architecture of an evolutionary fuzzer has only been slightly modified to introduce the notion of the program graph structure, see Figure 3.1. However, the implementations of NextSeed and CalculateEnergy have been significantly modified to adapt to the behavior in SpareFuzz. The reasoning behind these additions and modifications are described in more detail in the following subsections.
3.1. Design

(a) Pseudo-code for the architecture of a typical evolutionary fuzzer

\[
\begin{align*}
Q &= \text{GetInitialSeeds}() \\
\text{repeat} & \quad \text{Q = GetInitialSeeds()} \\
& \quad S = \text{NextSeed}(Q) \\
& \quad E = \text{CalculateEnergy}(S) \\
& \quad \text{for } i \text{ from } 1 \text{ to } E \text{ do} \\
& \quad \quad M = \text{Mutate}(S) \\
& \quad \quad O = \text{ExecuteProgram}(M) \\
& \quad \quad \text{if IsCrash}(O) \text{ then} \\
& \quad \quad \quad \text{Save } M \\
& \quad \quad \text{else if IsInteresting}(O) \text{ then} \\
& \quad \quad \quad \text{Add } M \text{ to } Q \\
& \quad \quad \text{end} \\
& \quad \text{end} \\
& \quad \text{until aborted by user} \\
\end{align*}
\]

(b) Pseudo-code for the architecture of SpareFuzz

\[
\begin{align*}
Q &= \text{GetInitialSeeds}() \\
Q &= \text{ReadProgramGraph}() \\
\text{repeat} & \quad \text{Q = ReadProgramGraph}() \\
& \quad S = \text{NextSeed}(Q) \\
& \quad E = \text{CalculateEnergy}(S, G) \\
& \quad \text{for } i \text{ from } 1 \text{ to } E \text{ do} \\
& \quad \quad M = \text{Mutate}(S) \\
& \quad \quad O = \text{ExecuteProgram}(M) \\
& \quad \quad \text{if IsCrash}(O) \text{ then} \\
& \quad \quad \quad \text{Save } M \\
& \quad \quad \text{else if IsInteresting}(O) \text{ then} \\
& \quad \quad \quad \text{Add } M \text{ to } Q \\
& \quad \quad \text{end} \\
& \quad \text{end} \\
& \quad \text{UpdateScores}(Q, G) \\
& \quad \text{until aborted by user} \\
\end{align*}
\]

Figure 3.1: Comparison of a traditional fuzzing architecture and the architecture in SpareFuzz. Additions are highlighted in green and modifications are highlighted in yellow.

3.1.3 Program graph structure

The feature of SpareFuzz which is perhaps most crucial for the implemented heuristics is the understanding of the program structure. SpareFuzz uses control-flow information extracted during compile-time for this purpose, which enables it to construct an in-memory representation of the entire program graph. The program graph contains the relationship between basic blocks, a separation between conditional branches and calls, and a look-up table used to quickly map branch identifiers to the program graph.

The relation between basic blocks along with the separation between conditional branches and calls is maintained by storing three collections per basic block: predecessors, intramodular successors, and called functions. The predecessor collection contains all basic blocks that immediately precedes the basic block in question. This includes both basic blocks within the same function, as well as basic blocks in calling functions. The collection of intramodular successors contains all possibly succeeding basic blocks that are present within the originating function. Since the execution continues along successors in this collection without returning, only one of these successors is visited every time the originating basic block is visited. Depending on if the basic block contains a conditional branch or not, the number of intramodular successors vary. Basic blocks implementing switch statements is a special case, where the number of successors is dependent on the switch cases. Other scenarios typically contain one or two intramodular successors.

On the contrary, the collection of called functions instead contains a list of branches that are always traversed when the basic block is visited. With this combination of the call graph (intermodular relations) and the control flow graph (intramodular relations), it becomes possible to reason about how execution through the program proceeds.

In order to calculate probability data within SpareFuzz, frequency data must be recorded and stored for later use. Since this must be done frequently, the graph is accompanied with a look-up table which helps speed up the process. This look-up table maps a unique identifier to every conditional branch. These identifiers are, in addition to being embedded in the program graph, also used in the instrumentation for tracking coverage. This way, a one-to-one mapping between the program graph and the coverage data is created.
3.1.4 Scoring system

The scoring system of SpareFuzz is to a large extent implemented with the help of probability data. This has been partly accomplished within fuzzing before, and Zhao et al. [11] propose a way to estimate future probabilities during fuzzing by using the rule of three from statistics. Their approach quantifies the difficulties of traversing unexplored branches but does so only for determining when a condition becomes too demanding for fuzzing to brute-force. Using this knowledge, they can dynamically apply a more sophisticated method for solving the condition symbolically. The probabilities of exploring an unexplored branch are modeled by multiplying the probabilities for traversing every branch leading to it. This is an efficient way of estimating the global difficulty of exploring the branch given that every collected seed is an equally suitable input. In SpareFuzz, the global difficulty is not of particular interest, but instead how likely specific seeds are to explore the branch. Therefore, the difficulty is sampled somewhat differently. The rule of three is still used to estimate the difficulty of unknown probabilities, but it is not multiplied with the probabilities of traversing the branches leading there. Instead, an estimate for the probability of reaching the node leading to the unexplored branch is used in its place.

Throughout SpareFuzz, probabilities for the occurrence of events are estimated using actual data whenever possible and using the rule of three when positive outcomes are missing. For the small number of cases when the rule of three cannot be used reliably, a constant value is used. Depending on the estimated probability, the value of this constant varies. This probability model is an extension to what is done by Zhao et al. which addresses the problem of not only considering how probable events that have not occurred yet are to occur, but also the probability of events that actually have occurred. In addition, their fuzzer DigFuzz can essentially ignore the requirement of a sufficiently large \( n \) because its absence will not lead to a probability that is low enough to be significant in their program logic. SpareFuzz, however, needs to deal with both of these cases, which is why they are addressed in the mathematical model. This model is shown in Equation (3.1), where \( p \) represents the number of positive outcomes, \( n \) the number of negative outcomes, \( t \) the total number of outcomes, and \( C \) the situation-specific constant. Within the context of SpareFuzz, positive outcomes indicate the occurrence of an event and negative outcomes indicate the absence of an event.

\[
P(\text{positive outcome}) = \begin{cases} 
\frac{p}{t} & \text{if } p > 0 \\
\frac{3}{n} & \text{if } p = 0 \text{ and } n > 30 \\
C & \text{otherwise}
\end{cases} \tag{3.1}
\]

For estimating the probability, and thus also the difficulty, of exploring an unexplored branch, two probabilities are introduced: \( \text{reachProbability} \) and \( \text{exploreProbability} \). The probability of exploring an unexplored branch is estimated by multiplying these two probabilities. More specifically, the combined value describes the probability of exploring a specific unexplored branch, given an input seed whose execution reaches the preceding node of this branch. We refer to the value as the \( \text{combinedProbability} \), see Equation (3.2).

\[
\text{combinedProbability} = \text{reachProbability} \times \text{exploreProbability} \tag{3.2}
\]

The \( \text{reachProbability} \) represents the probability of still reaching a specific basic block after mutating a given seed. This measure is introduced because fuzzing by design alters the path taken by a given seed, and it is therefore necessary to know how often individual conditions are still visited. The probability is calculated using Equation (3.1) where every time the seed has been mutated represents an outcome. Positive outcomes are achieved when the node is reached during execution with the mutated seed as input. The constant \( C \) is set to 1 in order to give an optimistic default. Values lower than 1 may cause the seed to never be selected, as
3.1. Design

similar seeds may be assigned higher values. Since none of these values are sampled globally, the reachProbability represents the probability for the specific seed it is calculated for. In contrast to the approach suggested by Zhao et al., this helps mitigate tendencies towards previously traversed paths. Consider a program with an early binary decision that continues in two different sub-trees. If the left branch of the binary decision contains many more interesting paths, the fuzzer will prioritize seeds leading to this sub-tree first. Since this causes the left branch to be taken more frequently than the right branch, the sampled probability of taking the right branch decreases. Mutations to a seed that covers the right branch might have a high probability of still covering this branch, but the global estimate would indicate that it is very difficult, regardless of which seed is used.

In another effort to combat tendencies towards suboptimal favoritism, branches with similar reachProbability values are treated equally. This is done to avoid favoring a few slightly better seeds when other seeds may be as beneficial for the fuzzing. This is accomplished by floor-ing the effort level, which is the inverse of the probability, and then using its corresponding probability. The mathematical definition of this mitigation is shown in Equation 3.3.

\[
\text{reachProbability} = \frac{1}{1 - \frac{1}{P(\text{reach})}} \tag{3.3}
\]

The exploreProbability is, contrary to the reachProbability, estimated using global data. The exploreProbability measures the probability of exploring a branch given that the preceding branch is reached. Every visit to the preceding branch represents an outcome. Since the probability is an estimate for unexplored nodes only, there are only negative outcomes. The constant C has a value of 0.01, which is chosen to represent an effort of 100 mutations when inverted. The motivation behind the globally estimated exploreProbability originates from two problems. First, the data sampled from a single seed is usually limited due to a low reachProbability, resulting in a small number of total outcomes. Secondly, a seed-local estimate would cancel out the reachProbability when combined into the combinedProbability. Consider the situation when the preceding node has been reached more than 30 times. The combination of the two probabilities would then be written as Equation 3.4, where the used notations have the same meaning as in Equation 3.1. Symbols with the subscript r represents values from the reachProbability and symbols with the subscript e represents values from the exploreProbability, note also that \( n_2 = p_1 \). The resulting value could be interpreted as the probability of exploring any unexplored branch when mutating the seed. This would assign the same probability to every unexplored branch in a seed, which represents a very improbable scenario.

\[
P = \frac{p_r}{l_r} \times \frac{3}{n_e} = \frac{3}{l_r} \times \frac{p_r}{l_r} = \frac{3}{l_r} \tag{3.4}
\]

SpareFuzz uses the introduced probabilities to improve the work by Gan et al. [10], based on the reasoning that every potential branch along the program path that a seed file gives rise to may not be as likely to explore. In the work by Gan et al. [10] it is shown that their algorithm for prioritizing seeds based on the number of unexplored neighboring branches alone (-br) slightly outperforms the additional consideration to how many descendent paths originate from these neighboring branches (-desc). These results on their own indicate that future work could benefit from being based on the algorithm -br. The algorithm -desc will, because of the path explosion problem, give a distribution of values along an exponential scale compared to the more linear distribution seen with -br. An unexplored branch near the beginning of the program would probably be assigned a very high value with the -desc algorithm while an unexplored branch further down the program logic would be assigned a smaller value. As the algorithm -br is unaffected by the program logic found beneath the branches, every
branch is assigned the same value. Intuitively, the distribution achieved with the algorithm -desc results in a somewhat unfair tendency towards certain paths. However, -desc is a better approximation of the benefit that the exploration results in, since the exploration of more paths is one of the goals of a grey-box fuzzer. The scoring system in SpareFuzz is based upon -desc because of this property, with the belief that the consideration to probabilities helps normalize the uneven distribution of scores caused by the path explosion problem.

The scoring system in SpareFuzz is realized with the help of a two-step approach. In the first step, the branchScore of every unexplored branch is calculated using the -desc algorithm. In essence, every unexplored branch is given a score equal to the number of unique intermodular paths beneath it. To limit this to a finite number, all cycles in the CFG are ignored. This value is combined with the exploreProbability to form a value that represents the potential reward given a certain effort or difficulty. Since this value represents the reward given a certain investment of time and energy, it is referred to as the branchROI or the Return On Investment for the branch. For every node that contains unexplored branches, a sum of the branchROI values is calculated, denoted by nodeROI. All of this data is stored within the in-memory program graph. This is done initially for the entire program graph. Whenever a new branch is explored, a partial update to the affected nodes is performed.

In the second step, a score is assigned to every seed by combining the nodeROI values computed previously with the reachProbability values. For every seed, every nodeROI value along the path which the seed traverses are first multiplied with their reachProbability and then summed together. This sum of "return on investment" scores is used to estimate the return on investment for the entire seed, which is denoted as the seedROI.

3.1.5 Seed prioritization

With the use of the scoring system described above, SpareFuzz prioritizes seeds based on their seedROI. This causes seeds that have a higher anticipated reward to be mutated first. Seeds with the same seedROI are ordered based on how many times they have been mutated. This is similar to how AFLFast partly prioritizes seeds. AFLFast partly prioritizes seeds based on how many times they have been selected from the seed queue, not how many times they have been mutated. Since the power scheduling in AFLFast assigns the same amount of energy to two seeds that have been chosen from the seed queue the same number of times, this is a solid strategy in their application. SpareFuzz does, however, use another method for power scheduling that in most cases assigns widely different amounts of energy to two different seeds. This difference in power scheduling design motivates SpareFuzz to prioritize seeds based on how much energy has been assigned to the seed in total, or in other words how many times they have been mutated.

The algorithm for seed prioritization, along with the calculation of the score used for prioritization, is outlined in Figure 3.2. Most undeclared functions in the pseudo-code have been previously introduced in the Scoring System subsection, while the expectedBranchEffort is introduced in the following subsection concerning the Power scheduling design. Note that the score calculation is optimized to run as rarely as possible in the actual implementation of this algorithm. One of the optimizations made is to only recalculate the numDescendantPaths value for basic blocks in paths that have recently been discovered. This is made possible by calculating this value for every basic block when starting the fuzzer, and then recalculating this value only for the directly adjacent blocks to the newly discovered paths, instead of for the entire program.
3.1. Design

function nextSeed()
    if seedList.empty?
        return initialSeed
    else
        return bestSeed()
    end
end

function bestSeed()
    best = seeds.first
    for seed in seedList do
        if score(seed) > score(best) OR
            score(seed) == score(best) AND
            seed.totalMutations < best.totalMutations
            then
            best = seed
        end
    end
    return best
end

function score(seed)
    score = 0
    for block in seed.traversedBasicBlocks do
        score += potentialScore(block) * reachProbability(block)
    end
    return score
end

function potentialScore(basicBlock)
    return sum( branchScore(branch) for every basicBlock.branches )
end

function branchScore(branch)
    return basicBlockScore(branch.to) / expectedBranchEffort(branch)
end

function basicBlockScore(basicBlock)
    if basicBlock.isVisited?
        return 0
    else
        return numDescendantPaths(basicBlock)
    end
end

Figure 3.2: Psuedo-code for the seed prioritization and corresponding score calculations
3.1.6 Power scheduling

The power scheduling in SpareFuzz is inspired by the model of the expected energy required to find new program paths proposed by Böhme et al. in AFLFast [8]. They introduce the expected value of the energy as \( E[X_{ij}] \) in Equation 3.5.

\[
E[X_{ij}] = p_{ij} + (E[X_{ij}] + 1)(1 - p_{ij})
\]

\[
= \frac{1}{p_{ij}}
\]

The expected value formalized above is a recursive version of the traditional definition of an expected value in statistics. The first summand \( p_{ij} \) represents the probability of discovering a new program path after the first mutation. The second summand represents the outcome of discovering a new program path after a later mutation multiplied with the probability of not discovering a new program path after the first mutation. This recursion can be simplified to a fraction, which indicates that the expected value is inversely proportional to the probability of discovering a new program path in exactly one mutation.

Even though Böhme et al. propose this model for calculating the expected value of the energy, they never use it within AFLFast. Their model is simply used to explain how the fuzzing behaves in theory. In their actual implementation, they use an entirely different heuristic that, in theory, should approximate the expected value quickly. Using sampled probability data SpareFuzz is instead able to implement their proposed model.

Furthermore, the model proposed in AFLFast models the probability of discovering any new program path. This probability is incompatible with the scoring system in SpareFuzz since only probabilities of discovering individual unexplored branches are tracked. A simple solution to this problem would be to use the highest probability, which gives an optimistic estimation of the probability of discovering any new program path. However, this might result in the assignment of suboptimal energy. It is possible that even more program paths are discovered given slightly higher energy. SpareFuzz attempts to estimate the optimal energy by considering all of the unexplored branches along the path that a seed traverses. Each unexplored branch is associated with a `branchScore` and an `exploreProbability`. Within the context of a specific seed, this probability is also multiplied with the `reachProbability` to form the `combinedProbability`. When calculating the energy for power scheduling, the `combinedProbability` is used in place of \( p_{ij} \) in Equation 3.5 to form the `expectedBranchEffort`, or \( E[X_{ij}] \).

The power scheduling in SpareFuzz is based on the assumption that an unexplored branch will be discovered after `expectedBranchEffort` number of mutations. However, the `branchROI`, or how many new program paths are expected to be revealed given a number of mutations, strictly diminishes as the number of mutations grows larger than the expected value. This is illustrated in Figure 3.3a, where every plotted value `branchROI_i` is calculated using Equation 3.6.

\[
branchROI_i(energy) = \begin{cases} 
\frac{branchScore_i}{energy} & \text{if } energy \geq expectedBranchEffort_i \\
0 & \text{otherwise}
\end{cases}
\]

Since every unexplored branch is considered during power scheduling in SpareFuzz, all plots in Figure 3.3a are ultimately summed, as illustrated in Figure 3.3b. The peak visualized in the figure is positioned at the energy that would theoretically result in the best return on investment. SpareFuzz performs the same summation of values internally during power scheduling to determine the peak value and its corresponding energy.
3.2 Implementation

SpareFuzz has been implemented as an in-process fuzzer, borrowing the mutation engine from AFL. Coverage tracking is added to the SUT during compile time via a custom LLVM compiler pass. The instrumentation responsible for keeping track of what branches are visited has been inspired by the work of Gan et al. [10] in an attempt to combat hash collisions. This section explains the rationale behind the various implementation decisions, together with the implementation details.

3.2.1 In-process fuzzing

The performance of a fuzzer is affected by many different factors, such as the generation of new input data, instrumentation overhead, and how fast the system under test can be executed. The execution speed of the system under test is mainly dependant on two factors: the time it takes to finish processing the given input, and the time it takes to restart the application. Of these two factors, the only one that can be influenced by the fuzzer is the time it takes to restart the application.

Ideally, the system under test is executed in sequence using the same input file for every run, but with varying content. This approach can then be scaled horizontally over processor cores, or even over multiple systems. However, to minimize the cost of scaling horizontally, the base performance must be sufficiently good. This is where the time it takes to restart the application comes into play because that is the overhead of executing an application in sequence.

There are a couple of approaches available for restarting an application in Windows: normal application starts, simulated application restarts, and function restarts. All of these approaches are described below and compared empirically in order to motivate the choice of the in-process fuzzing approach used in SpareFuzz.

Normal application restarts

The typical approach for starting an application is to request the start from the operating system. This involves a long series of actions in which system memory is allocated and configured for running the application. This complicated procedure is rather expensive from a time perspective and depending on the system configuration, anti-virus applications may choose to search through the application that is about to start for potential threats before it
is allowed to launch. While this slows down the procedure even further, it can easily be mitigated by disabling the anti-virus software.

On the other hand, a normal application start is easy to implement in basically all operating systems, and so is waiting for it to terminate. The behavior is mostly consistent across different system configurations, even if the speed may differ.

**Simulated application restarts**

The inherently slow nature of the traditional approach to starting, running, and waiting for an application has motivated other approaches that aim to simulate the same behavior. One such example is the approach used by the Windows fork of AFL, WinAFL [42], in which the program functionality is restarted without requiring a process restart by the operating system. The fuzzing framework does this by using DynamoRIO instrumentation to insert code which restarts the main function of the system under test every time it has completed.

While this approach allows for much faster execution than relying on the operating system, it is not without problems. Programs are usually not designed to be executed multiple times, and the introduction of such a feature may cause strange bugs to appear, leading to the appearance of false positives while fuzzing. Everything in a tested program that modifies or uses a global state is prone to potential problems, since this state is rarely reset on program exit. This is also an issue when testing using function restarts, but in that case the user can decide on the portion of the program that will be tested instead of being limited to the entire program. Furthermore, the available tools for instrumenting an application for simulated application restarts this are not all guaranteed to work on every system configuration, and updates to the operating system may sometimes cause them to stop working. Another issue with these tools is that they are may conflict with other invasive software, such as anti-virus applications.

**Function restarts**

For some systems under test, it is possible to circumvent the entire problem of quickly restarting an application. If the tested system can be executed from a single function, it is possible to run this function repeatedly. This effectively removes all the complexity of the simulated application restarts but maintains most of the benefits. The main issue with this approach is that applications with a global state may malfunction when tested using this approach. Another disadvantage is the inevitable requirement to write custom code that is only needed for making the application compatible with fuzz testing.

Depending on how the system under test has been designed, it is sometimes possible to test the application without writing any data to disk, which significantly improves speed.

**Comparison**

The comparison of the three approaches was structured as follows. For every approach, a test application was created, all originating from the following set of standard actions: first, a memory-mapped file is created on disk. This file is mapped to a memory segment within the test application, which enables it to be modified quickly as part of the fuzzing. After this, code that is specific to the approach in question takes care of running the system under test a number of times. The system under test was created for comparison purposes and only opens and reads the file it is presented with. For every new execution, the fuzzed input file is modified. The time it takes to complete all of the runs is measured with millisecond precision. Lastly, the memory-mapped file is closed, and the results are documented. Because of limitations posed by the different approaches, all of the tests were executed in a Hyper-V virtual machine running the official Windows 10 dev environment image.
### Table 3.1: Measured execution time for the presented approaches

<table>
<thead>
<tr>
<th></th>
<th>Normal restarts</th>
<th>Simulated restarts</th>
<th>Function restarts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>18 691</td>
<td>8 594</td>
<td>7 911</td>
</tr>
<tr>
<td>Runs</td>
<td>1 000</td>
<td>100 000</td>
<td>100 000</td>
</tr>
<tr>
<td>Runs per second</td>
<td>54</td>
<td>11 636</td>
<td>12 641</td>
</tr>
</tbody>
</table>

It should be noted that the memory-mapped files containing test data were only used to compare these different approaches. The input files containing test data in SpareFuzz are ordinary files. Memory-mapped files may enable quicker modifications of input data, but they are not without issues. Once a memory-mapped file has been created there is no way to shrink or enlarge it without recreating it entirely. The mutations made to input seeds during fuzzing typically modifies the size of the original data, but there is usually no way of instructing a tested program to only read part of the available data. This means that a memory-mapped file would always be read to the end, causing all sorts of issues for file formats that cannot be padded with, for example, whitespace. The usage of memory-mapped files would also introduce an upper limit for how large input files can grow.

The results from the comparison runs are shown in table 3.1. The number of runs for the approach using normal application restarts has been limited to preserve time during the evaluation, but it is unlikely that additional runs would influence the result enough to make a significant difference. The difference between the two other approaches is more interesting and shows that testing a function instead of simulating an application restart performs about 9% better.

During the measurement of data from the different approaches, it was found that the instrumentation framework DynamoRIO used for the simulated restarts was difficult to work with. Not only did it fail to work at all when third-party anti-virus software was running, but it also stopped working when attempting to use parts of the C++ standard library within the system under test. It is possible that other instrumentation tools that are developed with commercial funding, such as Intel’s Pin, might work better but there will always be some overhead introduced as a result of the dynamic instrumentation. There is also a risk that upcoming Windows updates break the instrumentation tool that is used.

The trouble of using instrumentation on different system configurations and the performance benefits of not restarting the application at all both motivates the use of function restarts within SpareFuzz. This limits the testing to applications and modules that does not maintain a global state, but on the other hand, it simplifies development and maintenance.

### Limitations of in-process fuzzing

When testing an application using in-process fuzzing, both the application and the code for testing the application resides in the same process. This is, however, problematic because fuzz testing aims to crash the system under test. A crash of the application will, because of the shared process space, also crash the test tool. This requires the presence of an external test oracle that can monitor the system under test for any crashes. While this is relatively simple for ordinary fuzzers, SpareFuzz requires knowledge of the system under test that can only be extracted within the same process. This limitation requires proper handling of the knowledge is communicated to the external test oracle even when a crash is imminent.

A simple way to share data between two processes in Windows is to use memory-mapped files. This allows for the mapping of a memory buffer within a program to either a file on disk or the system page file. SpareFuzz uses memory-mapped files to combat the problem of potentially lost data caused by application crashes by utilizing a standard operating system.
mechanism. File data that has been written by a program but not yet committed to disk is referred to as dirty content. This dirty content of a file is what would appear to be problematic during application crashes. However, the operating system is responsible for closing any open file handles every time an application terminates, both normally or abnormally. Chen explains in a blog post at the Microsoft blog *The New Old Thing* that there would have to be many special cases implemented within the kernel code of Windows to not write this dirty content to disk when an application does not close a memory-mapped file itself [44]. The most convenient way for both applications using memory-mapped files and for the kernel handling memory-mapped files is to persist data regardless. This property is heavily used within SpareFuzz because it allows the fuzzer to store data in memory buffers that are guaranteed to persist on disk even if the system under test crashes.

### 3.2.2 Implementation architecture

Originating from the limitations of in-process fuzzing, SpareFuzz is implemented as a dual application tool, with an in-process test application and an external test oracle that monitors the test application. The in-process test application, hereby referred to as the test runner, is a rather simple application that mainly generates mutated test cases, records coverage, and forwards all data to the test oracle. The test oracle is a more informed and complex application that implements the seed prioritization and power scheduling strategies that makes SpareFuzz unique. Every time the test runner is started or stopped, the test oracle prepares and processes data that is persisted within the oracle memory.

While this architecture is required for long-lived in-process fuzzing, it opens up for the possibility to process data only while the test runner is busy in order to optimize performance. Another benefit of this architecture is that multiple parallel test runners can be executed simultaneously.

The test runner and oracle communicate almost exclusively using memory-mapped files, except for the input seed file, which is an ordinary file. The oracle instructs the test runner how many times the seed file should be mutated before the oracle chooses a new one, and what seed file to mutate. Together with this information, a memory-mapped coverage file is also passed to the test runner. This coverage file keeps track of how many times every branch has been visited at most during a single run. A mutated input is deemed interesting if the previous number of visits is exceeded for any branch during execution. Every interesting input is then stored as a new seed file together with the coverage recorded during that run. In contrast to the fuzzer AFL, SpareFuzz does not perform any bucketing of visit counts for determining if a mutated input is interesting or not. When the test runner finishes, the oracle goes through every new input seed file and stores them internally together with their coverage profile. At this time, the oracle also reads another memory-mapped file containing the total number of visits for every branch accumulated over all mutations made to the chosen seed. This data is stored within the in-memory program graph and used to sample probabilities.

Alongside the dual application tool that is responsible for executing the fuzzing campaign, SpareFuzz also consists of a compiler optimization pass written for the compiler framework LLVM. This optimization pass instruments the compiled application with coverage tracking code that communicates with the test runner during execution. Since the data communicated is critical for the coverage-driven approach used in SpareFuzz, all applications tested with it must first be compiled using a compiler that contains this optimization pass.

### 3.2.3 Mutation engine

Since the generation of proper mutations from input seed files is a non-trivial task, the test runner uses the mutation engine from AFL. The fuzzer AFL uses many clever tricks that have been proven effective for fuzzing various applications. These tricks are both difficult
and unnecessary to re-implement, so SpareFuzz borrows code from the AFL project for this
single component. However, because the mutation engine in AFL is so tightly coupled with
the rest of AFL’s functionality, the mutation engine is not as powerful as it would otherwise
be. In particular, the coverage information that AFL collects is normally used to guide fur-
ther mutations. Since SpareFuzz does not share any coverage data with the mutation engine,
these mechanisms have been disabled. While this is indeed not optimal, it does not affect the
effectiveness of SpareFuzz significantly. This is due to the fact that SpareFuzz only utilizes
non-deterministic mutations during its execution. AFL resorts to deterministic mutations the
first time a new input seed is chosen for mutation, in which a pre-configured set of mutations
are made on the input data. This is only done once because the same mutations are made
every time, but using them is an effective way to find certain kind of bugs. SpareFuzz needs
to keep track of how many times every input seed is mutated, which becomes more difficult
with deterministic fuzzing since the mutation engine itself controls the number of mutations.
The inability to control the mutation count also conflicts with the new strategies in that Spare-
Fuzz that this thesis work aims to evaluate and has the risk of potentially skewing the results.

3.2.4 Coverage tracking

Coverage tracking in SpareFuzz is implemented using a compiler optimization pass written
for the compiler framework LLVM, which makes it possible to use it in the compiler Clang.
The compiler framework LLVM exposes the structure of the compiled module to the optim-
ization passes, which greatly simplifies the analysis required for coverage tracking. The
coverage tracking is implemented according to the recommendations made by Gan et al. in
order to mitigate hash collisions \[10\]. Within SpareFuzz, there are a few exceptions to the
recommendations made by Gan et al. First of all, the authors’ recommendation to enlarge
the coverage bitmap for large programs is ignored for complexity reasons which limit the
branch count of any tested application to 65,535 branches. Secondly, the coverage tracking in
SpareFuzz takes special care to only track intramodular branches, completely ignoring transi-
tions between nodes residing in different functions. This is accomplished by setting a special
flag before function calls that cause the next transition to be ignored. In addition, another
set of instructions are added after every function call so that the next transition is correctly
recorded.

Both the work of Gan et al. and SpareFuzz tracks coverage by keeping track of what branch
transitions are made, which requires globally unique identifiers. SpareFuzz fulfills this re-
quirement by running the custom optimization pass during the link-time optimization step.
At this point during the compilation, the entire compiled application is treated as a single
module. When running an optimization pass during another step, the compiler treats every
single source file as a separate module which creates issues with successfully establishing
globally unique identifiers. However, running an optimization pass during link-time opti-
mization is only possible by creating a fork of LLVM. This creates some issues with keeping
the forked compiler up-to-date, and an external plugin is usually recommended for this rea-
on. On the other hand, Clang and LLVM have no support for plugins on the Windows
platform, making forking of the compiler an inevitable consequence.
4 Evaluation

The evaluation SpareFuzz is based on data collected when testing a subset of the CGC test suite using two different fuzzing configurations. The test programs included in this subset are listed below:

- CGC_Hangman_Game (EAGLE00005)
- Character_Statistics (YAN0100003)
- Multipass (KPRCA00015)
- Multipass2 (NRFIN00010)
- Palindrome (CADET00001)
- root64_and_parcour (KPRCA00001)
- Sample_Shipgame (YAN0100001)
- Tennis_Ball_Motion_Calculator (YAN0100002)

The CGC test suite was created for a unique operating system named DECREE but has since been ported to Linux, macOS and Windows [24]. SpareFuzz is evaluated using the Windows port of the CGC test suite on two Google Cloud Virtual Machines, each running a subset of the tested programs:

- First five test programs: n1-standard-64, 64 virtual cores, 240 GB RAM
- Last three test programs: n1-standard-32, 32 virtual cores, 120 GB RAM

For every program in the test suite, two alternative configurations are evaluated against one another, each running five times for 24 hours per run. All test runs are seeded with a file containing the entire alphabet in upper and lowercase versions together with the numbers

1 Specifications for every machine type is available at https://cloud.google.com/compute/docs/machine-types
zero to nine. In addition to this, a second version of EAGLE00005 (CGC_Hangman_Game) is tested with an input seed that also contains the phrase “HANGEMHIGH!” at the beginning of the file. This addition is included to get past a password prompt that the user is greeted with when the program is started. In total, 90 test runs of 24 hours each are performed, which corresponds to 2,160 CPU hours of testing.

The two configurations are identified as the new heuristics and the old heuristics. The new heuristics identify an implementation of what is proposed in SpareFuzz, that is, a fuzzer that uses probabilities for seed prioritization and power scheduling. The old heuristics instead implement a probability-agnostic approach similar to what has been proposed in previous work by Gan et al. and Böhme et al. [10, 8]. This is achieved by assigning the same constant probability throughout the fuzzing campaign instead of calculating individual probabilities on the fly. In essence, the value 1 is returned every time a probability should have been calculated, making every event seem just as likely as every other.

The results are analyzed for statistical significance in order to determine if anything meaningful can be claimed from them. This decision, along with many other considerations made in the method, are all inspired from recommendations made by Klees et al. in their paper reviewing evaluation of fuzz testing tools [22].

4.1 Performance numbers

During the evaluation, the coverage over time is recorded along with every run. In addition, every found crash is logged. This information is supplemented with the initial burst and the peak of the coverage data. This information can be deduced from the coverage-over-time data and illustrates two important properties in a fuzzer more clearly. The importance of the initial burst is motivated by a desire to quickly cover a large part of the program, with the intuition that this benefits fuzzing. In addition, a similar metric has been used by Klees et al. [22]. This is further motivated in the “Discussion” chapter.

4.2 Power scheduling deviations

The power scheduling results are gathered by logging relevant information every time a new branch is explored. This information includes the estimated energy required to explore the branch, the actual energy required, and a probability-agnostic estimation made using the algorithm proposed by Böhme et al. [8].

The power scheduling deviations help answer how accurate the energy estimations made by the fuzzer are. For this metric, it is important to remember that a fuzzer estimates how many additional mutations are required before new coverage is achieved. If one heuristic estimates the required energy to be equal to 100 additional mutations, and new coverage is found after 75 mutations, the deviation is equal to 25 mutations.

In order to give a fair evaluation of the power scheduling deviations that is independent of other heuristics or eventualities that may be caused by the inevitable randomness of fuzzing, data for both heuristics is sampled from the same run. This is possible because the power scheduling algorithm proposed by Böhme et al. [8] is deterministic, and independent of any fuzzer state except for the number of mutations previously assigned to the current seed. If the fuzzer finds new coverage after \( x \) number of total mutations, then usage of the algorithm proposed by Böhme et al. would lead to discovery of new coverage after \( v = y + z \) mutations, where \( y \) is an accumulated number of previously assigned mutations, and \( z \) is the additional number of mutations estimated by the algorithm. The determinism ensures that every unique value \( x \) has a corresponding \( v \).
4.3 Results

The relation between a total number of mutations at which new coverage was found $x$, and the only possible way to reach at least this number of mutations using the algorithm by Böhme et al. makes it plausible to evaluate the deviations by augmenting such values. The evaluation data is gathered by running tests using the new heuristics and augmenting the values that would otherwise be estimated with the old heuristics. Doing so makes the deviations directly comparable without any influence from the seed prioritization or fuzzer eventualities, under the assumption that the augmentation of estimated energies is representative of the normal behavior.

4.3.1 Performance numbers

The performance numbers, or how many bugs were found and how much coverage was achieved during testing, is presented in this section. For each test program, two data sets are presented: the time to discovery for the bugs embedded within the test programs and the coverage over time.

Time to discovery

The time-to-discovery data are presented in Table 4.1 and 4.2. Test runs without any found bugs are distinguished by an empty cell in the table. The times are presented in the format hh:mm.

During evaluation it was discovered that Multipass had an unintentional bug which made the program crash on the first few seeds. These crashes have been excluded since they are out of scope for this evaluation.

Coverage over time

The coverage over time data is presented in graph format with both of the heuristics present in the same graph. Different runs using the same heuristics are distinguished by different line styles (dots, dashes, etc). Runs using the new heuristics for seed prioritization and power scheduling is colored in blue while runs using the old heuristics are colored in red. The data is presented in Figure 4.19 - 4.27. The peak coverage is presented in Table 4.3 and 4.4. In addition, the distribution of the area under curve for every heuristic and test program is presented in Figure 4.10 - 4.18.

4.3.2 Power scheduling deviations

The power scheduling deviations are presented as box and whiskers plots to give a clearer representation of the data distribution. The results for every tested program is presented in separate graphs found in Figure 4.19 - 4.27.
### 4.3. Results

Table 4.1: Time to discovery for the bugs found using the new heuristics (hh:mm)

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGC_Hangman_Game</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGC_Hangman_Game_Seeded</td>
<td>01:24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character_Statistics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multipass</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multipass2</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>Palindrome</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>root64_and_parcour</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample_Shipgame</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tennis_Ball_Motion_Calculator</td>
<td>00:00</td>
<td>00:00</td>
<td>00:06</td>
<td>00:00</td>
<td>00:00</td>
</tr>
</tbody>
</table>

Table 4.2: Time to discovery for the bugs found using the old heuristics (hh:mm)

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGC_Hangman_Game</td>
<td>14:58</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CGC_Hangman_Game_Seeded</td>
<td>01:40</td>
<td>14:20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character_Statistics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multipass</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multipass2</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
</tr>
<tr>
<td>Palindrome</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>root64_and_parcour</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample_Shipgame</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tennis_Ball_Motion_Calculator</td>
<td>00:38</td>
<td>00:00</td>
<td>00:00</td>
<td>00:00</td>
<td>00:05</td>
</tr>
</tbody>
</table>

Figure 4.1: Coverage data for CGC_Hangman_Game
4.3. Results

Figure 4.2: Coverage data for pre-seeded CGC_Hangman_Game

Figure 4.3: Coverage data for Character_Statistics
4.3. Results

Figure 4.4: Coverage data for Multipass

Figure 4.5: Coverage data for Multipass2
4.3. Results

Figure 4.6: Coverage data for Palindrome

Figure 4.7: Coverage data for root64_and_parcour
4.3. Results

Figure 4.8: Coverage data for Sample_Shipgame

Figure 4.9: Coverage data for Tennis_Ball_Motion_Calculator

Figure 4.10: Area under curve distribution for CGC_Hangman_Game

Figure 4.11: Area under curve distribution for pre-seeded CGC_Hangman_Game
4.3. Results

Table 4.3: Peak coverage achieved using the new heuristics

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGC_Hangman_Game</td>
<td>46</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>46</td>
<td>116.2</td>
</tr>
<tr>
<td>CGC_Hangman_Game_Seeded</td>
<td>176</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>163</td>
<td>165.6</td>
</tr>
<tr>
<td>Character_Statistics</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93.0</td>
</tr>
<tr>
<td>Multipass</td>
<td>108</td>
<td>106</td>
<td>111</td>
<td>108</td>
<td>111</td>
<td>108.8</td>
</tr>
<tr>
<td>Multipass2</td>
<td>197</td>
<td>193</td>
<td>197</td>
<td>195</td>
<td>197</td>
<td>195.8</td>
</tr>
<tr>
<td>Palindrome</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66.0</td>
</tr>
<tr>
<td>root64_and_parcour</td>
<td>86</td>
<td>85</td>
<td>91</td>
<td>85</td>
<td>86</td>
<td>86.6</td>
</tr>
<tr>
<td>Sample_Shipgame</td>
<td>369</td>
<td>373</td>
<td>373</td>
<td>373</td>
<td>344</td>
<td>366.4</td>
</tr>
<tr>
<td>Tennis_Ball_Motion_Calculator</td>
<td>171</td>
<td>171</td>
<td>171</td>
<td>171</td>
<td>171</td>
<td>171</td>
</tr>
</tbody>
</table>

Table 4.4: Peak coverage achieved using the old heuristics

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGC_Hangman_Game</td>
<td>176</td>
<td>163</td>
<td>165</td>
<td>165</td>
<td>46</td>
<td>143.0</td>
</tr>
<tr>
<td>CGC_Hangman_Game_Seeded</td>
<td>163</td>
<td>176</td>
<td>163</td>
<td>178</td>
<td>163</td>
<td>168.0</td>
</tr>
<tr>
<td>Character_Statistics</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93.0</td>
</tr>
<tr>
<td>Multipass</td>
<td>108</td>
<td>103</td>
<td>113</td>
<td>108</td>
<td>106</td>
<td>107.6</td>
</tr>
<tr>
<td>Multipass2</td>
<td>197</td>
<td>197</td>
<td>197</td>
<td>197</td>
<td>197</td>
<td>197.0</td>
</tr>
<tr>
<td>Palindrome</td>
<td>66</td>
<td>66</td>
<td>70</td>
<td>66</td>
<td>66</td>
<td>66.8</td>
</tr>
<tr>
<td>root64_and_parcour</td>
<td>84</td>
<td>114</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>90.6</td>
</tr>
<tr>
<td>Sample_Shipgame</td>
<td>219</td>
<td>370</td>
<td>358</td>
<td>210</td>
<td>219</td>
<td>275.2</td>
</tr>
<tr>
<td>Tennis_Ball_Motion_Calculator</td>
<td>169</td>
<td>169</td>
<td>169</td>
<td>169</td>
<td>169</td>
<td>169</td>
</tr>
</tbody>
</table>

Figure 4.12: Area under curve distribution for Character_Statistics

Figure 4.13: Area under curve distribution for Multipass

Figure 4.14: Area under curve distribution for Multipass2

Figure 4.15: Area under curve distribution for Palindrome
4.3. Results

Figure 4.16: Area under curve distribution for root64_and_parcour

Figure 4.17: Area under curve distribution for Sample_Shipgame

Figure 4.18: Area under curve distribution for Tennis_Ball_Motion_Calculator

Figure 4.19: Deviation data for CGC_Hangman_Game

Figure 4.20: Deviation data for pre-seeded CGC_Hangman_Game
4.3. Results

Figure 4.21: Deviation data for Character_Statistics

Figure 4.22: Deviation data for Multipass

Figure 4.23: Deviation data for Multipass2

Figure 4.24: Deviation data for Palindrome

Figure 4.25: Deviation data for root64_and_parcour

Figure 4.26: Deviation data for Sample_Shipgame
Figure 4.27: Deviation data for Tennis_Ball_Motion_Calculator
5 Discussion

The outcome of this thesis work has multiple dimensions that each deserves to be discussed in some detail. First and foremost, there is a need to interpret the results from the evaluation, but this only gives one perspective of the outcome. In order to fully understand why SpareFuzz behaves the way it does, both the design and implementation are also discussed and criticized. There is also a section dedicated to source criticism. Lastly, the decisions, benefits, and downsides of the evaluation are discussed.

5.1 Results

In general, the performance demonstrated in the results shows that the new heuristics introduced in SpareFuzz performs similarly to the base design it is compared to. There are, however, some aspects in which the two heuristics differentiate themselves from one another. These aspects are discussed in their respective section below.

5.1.1 Time to discovery

The time-to-discovery data is mostly indicating the difficulty of finding bugs within the CGC dataset when using a coverage-guided grey-box fuzzer. Most of the bugs were never discovered by any of the heuristics. Out of the bugs that were found, the ones within Palindrome and Tennis_Ball_Motion_Calculator are very shallow and were found within minutes or seconds. The bug embedded in Tennis_Ball_Motion_Calculator was almost exclusively discovered by SpareFuzz first, but with the few number of tests, it is difficult to say if this is a coincidence or actually representative of any superior performance. This uncertainty is aggravated by the shallowness of the bug, as the difference in time to discovery is smaller than it otherwise might have been.

The only program in the test suite that is representative of any difference in time to discovery is CGC_Hangman_Game. The results from both the original and the pre-seeded version are more interesting as they actually display a clearer difference between the two heuristics. Despite this, it is difficult to say which version performs best. Based on the average time to discovery values, the new heuristics perform better on the pre-seeded version, but when
5.1. Results

comparing the number of runs where at least one bug was found, the old heuristics instead perform better for both versions.

It should also be noted that this comparison of time to discovery is slightly different from comparisons of effectiveness in other fuzzing works. Böhme et al. do, for example, measure the number of unique crashes over time [8]. In addition, Klees et al. give a recommendation to always present performance over time when evaluating tools for fuzz testing [22]. The evaluation of SpareFuzz does not present any such data because it is performed on a limited test set. The chosen CGC test set contains only one or (at most) two bugs per program, while other test sets might contain hundreds or even thousands of unique bugs. This limitation makes the performance over time meaningless.

5.1.2 Coverage over time

Because of the differences between various runs, the graphs presenting the coverage over time is difficult to interpret without a proper analysis of the results. There are, of course, many different aspects in which the coverage over time can differ, but this analysis is limited to two key factors: the initial burst and the peak coverage.

The initial burst represents how quickly a base-line coverage is achieved, with the intuition that a quicker burst is better for the fuzzing campaign. Klees et al. quantify a value that approximate a similar property using an area under curve metric for the number of found bugs [22]. Since the evaluation of SpareFuzz contains very few bugs, and thus lacks a record of the number of found bugs over time, such a value cannot be calculated. Instead, a value based on the coverage over time is used. This is not very representative of the fuzzer effectiveness since runs with medium-to-high coverage but no crashes will be assigned high numbers. However, it gives an indication of the ability to quickly cover a more substantial part of the tested program.

Of all the 8 tested applications, SpareFuzz covers a larger area under curve in half of them. Out of the remaining applications, SpareFuzz performs worse in 3 cases and equal in 1 case. Note that the non-seeded and pre-seeded versions of CGC_Hangman.Game are interpreted as the same tested application.

It appears that the area-under-curve results correlate mildly with the time to discovery. The time to discovery for the program Tennis_Ball_Motion_Calculator is somewhat shorter for SpareFuzz, and the area-under-curve is also somewhat larger. The same cannot, however, be said about the program Palindrome, but when observing the graph in Figure 4.6 it becomes clear that the additional coverage found using the old heuristics was found later than the bug was discovered. Because the bug was found so early, it makes the relation between the time to discovery and area under curve more difficult to analyze. The program itself is also rather simple and in retrospect not very representative for comparing fuzzers against each other.

The peak coverage presented in Table 4.3 and 4.4 shows how much coverage is achieved in the different runs. The average of these peaks is an important metric because it shows how well the fuzzer is able to consistently cover many program states, regardless of if it is early or late into the fuzzing campaign. In this comparison, SpareFuzz performs better in roughly the same 4 test programs as in the area-under-curve comparison, with the exception of root64_and_parcour. It also performs worse and equal in the same test programs, again with the exception of root64_and_parcour. For the individual runs, SpareFuzz peaks highest for 2 of the tested programs, while the version using old heuristics peaks highest 4 times. The results indicate that SpareFuzz performs better more consistently with respect to achieved coverage, but that the peak coverage may be slightly better in some runs using old heuristics.

\[^1\]The test program root64_and_parcour has an outlier value recorded during the second run using old heuristics. When this outlier is ignored, the new heuristics has a slightly better mean peak coverage.
5.2 Design

5.1.3 Power scheduling deviations

The power scheduling deviation results are difficult to draw any meaningful conclusion from. It appears that the median deviation is better for 3 of the tested programs when using the new heuristics. In some cases, the distribution of the deviation data is very similar between the two heuristics, while it is different in other cases. This could indicate that some of the tested programs benefit from having a probability sourced power scheduling, while some of the tested programs are tested better with the more traditional implementation proposed by Böhme et al. [8]. The test programs where the deviation results differ the least are also the test programs where a more steady increase in coverage is present. This might be a hint that the power scheduling is less important when new coverage is frequently detected, but it is not possible to make any conclusion from such a small data set. However, such a correlation is not seen between the power scheduling deviations and the coverage over time or number of bugs found.

Another important takeaway from the results and the way they were collected is the fact that the energies used to calculate deviations for the old heuristics are sampled using data from runs using new heuristics. This is possible thanks to the old energy calculating being deterministic which also allows for comparisons against the same number of actual mutations required. However, this way of collecting data might lead to skewed results. The two approaches behave differently in other aspects and in addition to this, assigning a different number of mutations might lead to other paths being taken, which could further affect the result.

The performance of SpareFuzz was never compared with the different enhancements in isolation because of time limitations, but such a comparison would be make it easier to evaluate the proposed energy assignment heuristic more accurately. Doing so would make it possible to compare performance in time to discovery and coverage over time instead of relying on energy deviations alone.

5.1.4 Statistical significance

The results from the evaluation of SpareFuzz are largely non-decisive, with no heuristic having a clear advantage over the other. In some cases, the new heuristic appears to perform slightly better, and in other cases, the old heuristics appear to perform better. The difference between these two heuristics is not significant enough to indicate any benefit of either heuristic over the other. However, the results may give hints on what benefit could be achieved. This dilemma applies to the results on time to discovery and coverage over time. It could be so that the enhancements introduced in SpareFuzz are more beneficial for certain types of programs, and then there would never be any statistical significance when considering a set of widely different programs together. There is currently, as noted by Klees et al. [22], a lack of proper test suites for evaluating fuzzers. If any such test suite is created, it should preferably contain a categorization that could help distinguish different program structures and properties. This information could then be used to separate results in order to prove statistical significance for a subset of all programs. The generality of fuzzers may hinder their performance, and a fuzzer better suited for the job it is assigned to should be preferred over a fuzzer that performs only sufficiently good for all applications.

5.2 Design

There are a couple of design decisions in SpareFuzz that deserves to be discussed, both fundamental difficulties in introducing probabilities to the field of fuzzing but also issues present in the field of fuzzing that might affect the effectiveness of SpareFuzz.
5.2. Design

5.2.1 Estimating meaningful probabilities

As mentioned in the “Design and Implementation” chapter, the rule of three from statistics conflicts with the sampled probabilities if used within the same context. In essence, the probability for still reaching a particular basic block after mutating an input seed (reachProbability) and the probability of exploring an unexplored branch (exploreProbability) cannot both be calculated using the same set of sampled data. Therefore, SpareFuzz strikes a compromise by using data sampled from a single seed to calculate the reachProbability and data sampled from all seeds to calculate the exploreProbability. This has the added benefit of having more data for calculating the exploreProbability. However, it is not guaranteed that doing so accurately represents reality. Some input seeds may be very different from the input that ultimately explores a particular branch even if they both reach the preceding node. Such input seeds could negatively impact other seeds that reach the same preceding node by decreasing the exploreProbability and thus also the combinedProbability used for prioritization and power scheduling.

Ideally, the exploreProbability would be based on data that is specific to every seed the combinedProbability is calculated for, but without conflicting with the reachProbability. This would also require that there is more data available to calculate a proper exploreProbability. Currently, a preceding node is rarely visited after mutating an input seed that itself visits this preceding node. Not only does this cause the estimation of the exploreProbability to be difficult to calculate, but it also results in low values of the reachProbability. Preferably, the reachProbability should be sufficiently high to avoid wasting time testing inputs that never reach the requested basic block.

Chen et al. identify that a proper granularity of mutations is important within the context of a directed grey-box fuzzer [21]. Depending on the current goal of their fuzzer, either finer or more coarse mutations are used. This decision is made from how different the path through the program needs to be. Finer mutations are more likely to preserve the current path while more coarse mutations usually result in a completely different path being taken through the program. For a directed fuzzer, the choice between performing a small or large alternation to the current path can usually be deduced from the distance between the current location and the requested location.

For a coverage-guided fuzzer like SpareFuzz, on the other hand, this choice becomes a lot more difficult. The main idea behind SpareFuzz is that seeds with many high-reward adjacent paths are prioritized. The branches leading to these adjacent paths may be located at different depths within the program and thus have different corresponding mutation granularities that are most likely to cause their condition to flip while preserving the value of all above conditions. Because there are many different mutation granularities that could result in the discovery of new program paths, it becomes difficult to reason about what would be the best granularity to use. Perhaps the used granularities should be weighted towards values that help discover branches with the potential of revealing the most paths (i.e., branches with the highest branchScore). The author behind SpareFuzz believes that more research must be conducted before anything certain can be claimed about this issue in coverage-guided grey-box fuzzing. Not only to investigate if it is suitable for increasing the reachProbability, but also for investigating if adjusting the granularity of mutations alone is precise enough to target specific branches along a path through the program.

Another potential issue with the probability estimations made in SpareFuzz is if the conflict between the reachProbability and the exploreProbability when calculated from the same set of data represents reality better than the current combinedProbability. As explained in the chapter “Design and Implementation”, the conflict between the two probabilities when based on the same set of data results in a value that describes the probability of exploring any previously undiscovered branch when mutating a seed. This is an unwanted consequence since much
of the intuition behind the design of SpareFuzz relies on the existence of differences in the difficulties of exploring unexplored branches along a program path. However, there might not be enough data available to estimate such probabilities accurately. It might be the case that the added `exploreProbability` is superfluous and that the `reachProbability` alone is a better alternative since it can be based on more reliable data. This effectively ignores the difficulty of actually breaking the condition that stops an unexplored branch from being explored but might be better than the alternative. Currently, the calculation of the `combinedProbability` uses a loop-hole in order to include an estimation of the difficulty of discovering an unexplored branch, but this estimation may be inaccurate and of little value. It may be possible to yield more accurate results using an estimation of the difficulty based on fundamental program analysis rather than sampled data, but the issue should be more thoroughly studied before anything can be concluded.

In close relation to the issue with the conflicting probabilities is the potentially incorrect application of the rule of three from statistics. The rule of three aims to give a confidence interval for the probability of the occurrence of an event which has not yet occurred. This confidence interval states that the probability of the occurrence is less than or equal to the estimated value with a 95% certainty. Zhao et al. uses the value estimated using the rule of three for detecting when the occurrence becomes too improbable [11]. SpareFuzz, on the other hand, is based on the assumption that the estimated probability is sufficiently close to the actual one. As time passes and more data is sampled, the estimated probability decreases until the corresponding unexplored branch is eventually discovered, but this method might not be the best possible alternative. Ideally, a more reliable method should be used for estimating the probability of exploring unexplored branches. Such estimations should either give approximations close to the actual values or give a lower limit to the value instead of an upper limit. Using lower limits better suits the heuristics used in SpareFuzz: choosing seeds with the highest probability of discovering new branches.

5.2.2 Benefits of using probabilities

The introduction of probabilities within fuzzing is motivated by the need for a mechanism that limits the time wasted trying to explore parts of the program that are either unexploitable or difficult to explore. Within coverage-guided grey-box fuzzing, it is favorable to have a large available search-space, both for having access to a larger space of exploitation but also to increase the potential of further expanding the search space. Probabilities help realize this goal by directing the fuzzing towards parts of the program that are most likely to result in the discovery of new program paths, and therefore a larger search-space. This advantage is dependent on the probability estimations being efficient enough. If the probabilities take too long to be estimated accurately, time may instead be wasted trying to uncover unexplored branches that are difficult to explore. However, if probabilities are accurate enough, the benefit they introduce should exceed that of heuristics that help avoid non-beneficial decisions in other fuzzers.

Considering that probabilities are accurately estimated within a reasonable time-frame; there are still some situations in which probabilities may not be the best alternative for quickly finding new coverage. Different program structures have motivated different approaches within fuzzing, and while grey-box fuzzing helps test code at a greater depth than black-box fuzzing, it still struggles with getting past very specific conditions. Testing of programs that either contains many such hard-to-guess conditions or are very linear in its control-structure (i.e., having a low degree of branch-out in the CFG) receive little benefit from the introduction of probabilities. Programs with some (but not many) hard-to-guess conditions may benefit from introduced probabilities since it would then result in better prioritization of other, more simple, paths through the program. However, it should be noted that this only hides the
5.2. Design

Problem. Most of the hard-to-guess conditions will probably never be explored, and thus a portion of the program may remain untested. Luckily, there exist tools that aim to make some hard-to-guess conditions such as string and number comparisons easier for a grey-box fuzzer to handle by splitting them into multiple smaller conditions, allowing the fuzzer to track more fine-grained increases of coverage. One of these tools is laf-intel \cite{45}, which splits comparisons such as strcmp and switch statements in order to make fuzzing easier. The use of these tools does, however, not simplify the issue of adequately tracking coverage within different iterations in a loop. Some comparisons are implemented either as loops or as functions that are called many times during the execution of a program. Chen and Chen \cite{13} proposes a solution to the problem of separating coverage in different calls to the same function by encoding the context within the coverage bitmap, but the distinction between different loop iterations is a more difficult problem to solve.

In linear programs, the problem instead lies in wasted benefit. Since the introduction of probabilities is useful for distinguishing between different alternatives, it is best suited within programs where the fuzzer needs to make decisions on what path to explore next. In linear programs, this problem does not exist to the same extent, and successful fuzzing is instead more dependent on quickly cycling through all relevant seeds and performing good mutations. Here the introduction of probabilities could instead hinder the success of fuzzing by spending more time than what would be necessary.

5.2.3 Exploration vs. exploitation

During the process of designing SpareFuzz, several challenges of a coverage-guided fuzzer were highlighted. Perhaps most importantly, a successful coverage-guided fuzzer must be able to do two things well: cover larger portions of the system under test and trigger any potential bugs in the discovered code. This explore vs. exploit tradeoff concerns the problem of allocating a limited set of resources to different options in order to maximize the expected gain, where the benefit of each option is unknown in advance but may be better understood when more resources are allocated to the option in question. The exploration phase is based around the task of determining how beneficial each option is, while the exploitation phase is concerned with using the collected information in order to maximize the gain. Within the context of a coverage-guided fuzzer, the exploration concerns the corresponding exploration of the program state space, while the exploitation is concerned with focusing on what has already been explored in order to find bugs.

The design of SpareFuzz is very focused on the task of covering a larger portion of the system under test, and the improvements to the fuzzer are focused around this. The probabilities used in SpareFuzz are mainly introduced to achieve better code coverage (i.e. exploration), and there is little to no thought in how this could help find bugs in already discovered code (i.e. exploitation). SpareFuzz is, due to its reliance on the \texttt{desc} scoring algorithm, closely related to the fuzzer CollAFL proposed by Gan et al. \cite{10}. In their work, Gan et al. do not discuss this issue, yet they claim that their introduction helps improve path discovery, crash finding, and vulnerability discovery. However, their lack of consideration to an exploitation phase that is also missing from SpareFuzz should result in worse vulnerability discovery. The reason why Gan et al. choose not to discuss this issue could be originating from how their fuzzer is implemented. CollAFL is based on the fuzzer AFL which has proven to be effective for discovering vulnerabilities, even though some argue that it has a suboptimal solution to the exploration vs. exploitation trade-off \cite{8}. Since SpareFuzz does not originate from AFL but is instead implemented from scratch (disregarding the borrowed mutation engine from AFL), it needs to solve the exploration vs. exploitation trade-off on its own. In its current design, SpareFuzz stops mutating a seed completely once it no longer has the potential of discovering new program paths. This means that such seeds are excluded from any co-
5.3. Implementation

Incidental exploit phase that could otherwise occur if the fuzzer tries to break unbreakable conditions and mutated the same set of seeds over and over.

SpareFuzz and other fuzzers not originating from an extension to an already established fuzzer should, therefore, take special care to avoid this scenario if used for finding vulnerabilities in real-world software. The author of this thesis work believes that the exploration vs. exploitation trade-off should be more closely studied within the context of a fuzzer that is already heavily focused on exploration. During the design of SpareFuzz, several proposals to a solution to this problem were considered but never used. One of the proposed solutions was to separate the execution into two separate processes with different purposes. One process would be responsible for exploring the program space, in essence performing all of the work that is currently done in SpareFuzz. The other process would try to even out how many times every part of the program was tested by considering all seeds to be of value. This could have been realized using metrics such as how many times each seed has been mutated, which is already tracked and available within SpareFuzz. However, because this addition had the risk of making the two evaluated designs less different, it was never used in the final design, but since this addition may be a simple mitigation to the exploration vs. exploitation trade-off, further research should be conducted on its benefits.

5.3 Implementation

While the implementation of SpareFuzz resulted in a well-functioning fuzzer, some implementation-specific choices deserve to be mentioned. This is done both for the sake of criticizing some aspects of SpareFuzz, but also for creating awareness around some of the limitations in the techniques that enabled the creation of SpareFuzz.

5.3.1 The short-lived test runner

While the dual-application architecture of SpareFuzz was a mandatory design decision caused by the choice to implement in-process fuzzing, how it was implemented could have been improved to increase the performance of SpareFuzz. In its current implementation, SpareFuzz starts one new process for the test runner every time a new seed is chosen for mutation instead of re-using the old process. This means that an overhead caused by starting and stopping the test runner, and thus also the tested application, is introduced within the fuzzer. The impact of this overhead is not very large in contrast to how long the test runner executes for, once the number of assigned mutation grows beyond the initial values. The threshold at which the time for starting and stopping the test runner becomes less significant is just a few seconds into fuzzing, which is why this issue was never resolved in SpareFuzz. However, the test runner is still short-lived in comparison to how long it could have been running for otherwise. There is thus some performance benefit to gain, especially when testing larger applications that may take even longer to start. Despite all of this, there are some benefits to the short-lived approach currently used. Since the test runner is short-lived, there is less need to worry about potential memory leaks causing any problems. These memory leaks could appear both within the test runner itself or within the tested program. While it may be interesting to discover memory leaks during fuzzing, as that is also an error in the tested application, such errors are more easily and efficiently found using memory analysis software such as Valgrind. Memory leaks within the test runner itself is most certainly an unwanted occurrence, but it is more important to focus on ensuring that the test oracle does not contain any memory leaks since it is a much more critical component of the fuzzer. If the test oracle crashes, the entire fuzzing campaign comes to an end.
5.3. Implementation

5.3.2 Instrumenting programs with LLVM

Within all coverage-guided fuzzers, there is a need to instrument the system under test with coverage tracking code. This can either be done during compile-time or using dynamic instrumentation for already compiled programs. SpareFuzz relies on instrumentation made during compile-time since it makes it easier to reason about the structure of the program, and the approach is compatible with more system configurations. However, this effectively limits the fuzzer to programs where the source code is available. This decision, along with the choice of using globally unique identifiers for tracking coverage, also creates complications concerning the scope of compilation since every tracked portion of the program must be compiled simultaneously. SpareFuzz aims to mitigate this issue by running the LLVM pass that instruments the code during the link-time optimization step. At this point, all parts of a compiled program are available to the instrumentation pass and global identifiers can be successfully established. What this mitigation does not solve is the problem of testing multi-modular programs, where the program logic is separated into multiple dynamic libraries (or DLLs on Windows). Since every application component on disk (e.g., every EXE and DLL) is compiled separately, they cannot be reasoned about together during compilation. Dynamic binary instrumentation, on the other hand, instruments applications once they have been loaded into memory when all libraries are available within the process memory space. This makes it possible to establish unique identifiers at the cost of more sophisticated tools for instrumentation and loss of compatibility. It is possible that unique identifiers could also be established with LLVM instrumentation using static libraries. When including other modules into the application as static libraries, the code of the libraries is embedded into the main executable instead of being available as separate files. This should make it possible to instrument all of the application components as a single unit.

5.3.3 Other potential improvements

While the implementation of SpareFuzz takes special care to use sufficiently large data types for storing positive integers, the handling of floating point numbers could be improved. SpareFuzz is implemented on Windows which limits the implementation to 64-bit double values (i.e. floating point numbers). Greater care could be taken to, for example, study how multiplication and division of these numbers is best performed with regards to the order of operations so that the least lossy mathematical operation is performed first. Another improvement would be to use a larger data type, but that requires the use of an external library. Regardless, the handling of floating point numbers could be improved, which should help improve reliability when running SpareFuzz for a longer time.

During testing, it was found that SpareFuzz tends to generate rather small seed files. This behavior, which comes from the mutation engine borrowed from AFL, might not always be beneficial. SpareFuzz only saves newly generated seeds that are deemed interesting, and if the first seed that traverses a unique path is small, the fuzzer could get stuck. Further mutations of this saved seed would in most of the cases result in a smaller or similarly sized seed. If a condition in the program requires a much larger seed size in order to be fulfilled, it is unlikely that new coverage originating from this condition is discovered. The reason for this behavior in the mutation engine of AFL is motivated by a need to maintain a stable system. If the seed size has a higher tendency to grow than to shrink, it would grow larger with time and eventually be too large for the fuzzer to process efficiently. However, the shrinking could be a large factor in why the performance of SpareFuzz is non-deterministic. It is possible that the changes made to the mutation engine before inclusion within SpareFuzz contribute to the behavior that is observed. In particular, the missing seed splicing and deterministic mutations otherwise used in AFL might have resulted in the generation of more large files. However, it is not certain that the issue lies in the mutation engine. Another possibility is that the limitation of one seed per program path is too strict. Regardless of what causes the issue,
5.4 Source criticism

The theory behind, and the inspiration for, SpareFuzz is sourced from a large number of peer-reviewed papers within the field of fuzz testing. Most of the papers that introduce key ideas used in SpareFuzz are published in recent years, making the design of SpareFuzz up-to-date with the current state of the field. However, the sources being peer-reviewed and recent does not guarantee their validity. When authors, along with their study, also make their implementation available, it is possible to validate that their results are accurate. When the implementation has not been made public, however, the results have the be trusted blindly.

It should be noted that no effort of validating previous research results has been made for the realization of SpareFuzz, so every presented result is trusted blindly to some degree. This is a potential threat to the validity of the design of SpareFuzz because the authors of inspiring papers might have chosen to exclude bad results from the published study. Even if the authors did not make any attempt of skewing the results, the evaluation itself might be involuntarily flawed, and according to Klees et al., most of the evaluations made within the field of fuzz testing are flawed in more or less critical ways [22]. This is inevitable until the evaluation quality within the field of fuzz testing improves, but it is indeed something that deserves to be mentioned.

Some information presented in this thesis is not from peer-reviewed papers but instead from blog posts and websites. This information has, however, been carefully selected to ensure that the content is trustworthy. All of the information has either been published by a reliable source, such as Microsoft, or cited in published papers.

5.5 Method

In retrospect, the evaluation of SpareFuzz was performed on a suboptimal set of test programs. Klees et al. recommend that evaluation of fuzzers is done using a test suite containing programs with easily distinguishable bugs [22]. Such programs appear in the artificial test suite LAVA-M and the DARPA Cyber Grand Challenge (CGC). Other real-world programs could be used as well but would require a more extensive analysis of the results in order to deduplicate bugs. Klees et al. make this recommendation because it is important to test the performance of fuzzers against ground truth, and they claim that the available methods for deduplicating crashes are not accurate enough. They also state that there is a lack of an established standard test suite meant to test the performance of fuzzers.

Given that the use of real-world programs was not reasonable within the available time-frame, the decision was made to use a test suite containing programs with easily distinguishable bugs. LAVA-M contains programs with many bugs embedded within them and originates from a modification to typical command-line applications used on Linux. The programs used during the DARPA Cyber Grand Challenge were instead small in size and artificially engineered for the challenge, in which competitors were tasked with creating automatic tools for vulnerability discovery and patching. The bugs hidden within these programs are non-trivial for a coverage-based fuzzer to find, but considering that the GCG test suite was the only one of these alternatives that was available on Windows, it was chosen for evaluating SpareFuzz. In hindsight, it is highly questionable if this was the correct choice. Since the difficulty in finding any bugs was very high, it is possible that the potential benefit of SpareFuzz compared to other alternatives was not shown. The structure of the applications in the CGC test suite is also very linear and contains few branches, making seed prioritization less beneficial. The problems with this structure have been discussed previously and, unfortunately,
SpareFuzz was tested on programs containing such structures. It might have been better to test SpareFuzz using real-world programs that do not have a pre-established ground truth just for the sake of comparing the fuzzer in a situation it was designed for. However, the evaluation made with the chosen test suite exemplifies the limitations of the fuzzer, which is also important.

5.6 The work in a wider context

Research on tools for automatically finding vulnerabilities in software becomes increasingly important as cyber warfare and threats against important infrastructure increases. By keeping automatic means of vulnerability discovery, such as fuzz testing, away from the general public, malicious actors have a potential advantage over software vendors that is unaware of potential issues in their software. By instead popularizing and utilizing these tools at a larger scale, both software vendors and malicious parties are equipped with the same tools. Since it is impossible to stop malicious parties from using the same knowledge as is available to the general public, the best way of minimizing their impact is to not let them have an advantage. It is, of course, possible that research on tools for automatic vulnerability discovery may end up in the hands of malicious actors before being properly implemented in tools used by software vendors. However, such a way of thinking must not hinder the research on new and more effective methods that can ultimately be used to increase security in the entire software industry.
The design of SpareFuzz was inspired by a lack of good heuristics for seed prioritization and power scheduling that adapts to the program behavior while still maintaining a low level of program analysis. SpareFuzz takes a novel approach to this problem by recording how frequently different parts of the system under test are executed during fuzzing and then using this information for guiding the fuzzer towards undiscovered code that has the highest probability of being discovered. With the help of this frequency data, SpareFuzz is also able to more accurately predict how many mutations are required for revealing new coverage.

While the design of SpareFuzz is simple at a high level, there is difficulty involved in estimating proper probabilities for events that have yet to occur. SpareFuzz uses the statistical rule of three for estimating these probabilities, but this formula only gives an upper bound of the probability. This upper bound is decreased as the number of samples used to calculate the value increases, but at a suboptimal pace. Future work on fuzzers that utilize probability data for guiding the fuzzing should focus on this issue in order to either propose a better formula or to determine if the rule of three is inappropriate in this context.

For both seed prioritization and power scheduling, SpareFuzz combines the probability of reaching a specific, already discovered, program location with the probability of discovering an unexplored location that is connected to this already discovered location with a single conditional branch. This is partly motivated by the limitations in the current estimations of unknown probabilities, but there is a more important reason for this decision. The mutations performed during fuzzing are designed to cause the program path traversed when executing a seed to change. This is a crucial property of fuzzing that makes the discovery of new program paths possible. However, when the goal is to explore a particular conditional branch, it is of little value if the conditional statement that precedes this branch is not executed. Therefore, SpareFuzz takes into account how frequently the preceding conditional statement is executed when originating from the same particular seed. This consideration is also made by Zhao et al. but accomplished by multiplying the probabilities of every node in the program path leading up to the conditional statement in question [11]. SpareFuzz instead computes unique probabilities for every seed, which is especially useful when multiple paths lead to the same conditional statement. The proposed solution for properly utilizing probabilities in SpareFuzz is a theoretically sound approach, but it is difficult to compare the real
benefits when the probability estimations may be suboptimal. There may be a hidden flaw in the theoretical reasoning that motivates the proposed solution. For this reason, future work should be conducted to compare the solution proposed in SpareFuzz with other solutions. These comparisons should be made using the same methods of probability sampling in order to compare only the actual application of this data.

In terms of effectiveness compared to a fuzzer similar to CollAFL, SpareFuzz does not seem to have any significant advantage. Fuzzer effectiveness is traditionally measured by the number of bugs found within a given timespan. SpareFuzz and CollAFL respectively find approximately the same number of bugs, and the time-to-discovery data is too uncertain to draw any conclusions from. In less common cases, the ability to cover most program states in the shortest time is used in order to extrapolate the performance. While this only mildly correlates with how effective a fuzzer is, SpareFuzz is compared using it because of the lack of proper data. The coverage data shows that SpareFuzz is marginally better at discovering new coverage than the compared fuzzer. There is a small difference in the number of cases in which SpareFuzz performs better in this regard, that may very well be a coincidence, but the coverage obtained using SpareFuzz is more consistent than the alternative. This indicates that the performance of SpareFuzz is more deterministic, something that is not very unexpected, considering that SpareFuzz is designed to better adapt to the program behavior. This property alone could motivate the use of SpareFuzz, considering that the performance difference is so small. However, it might be wiser to compare performance on candidate programs prior to full-scale testing, as there is an apparent difference in performance depending on how the program is structured.

While the assigned energy deviates less consistently from the energy actually required to reveal new coverage, compared to the alternative, the results on power scheduling indicates that program structure could have an impact on what implementation is the most beneficial. It appears that the proposed power scheduling in SpareFuzz provides no obvious benefit for the general case, but the results are inconsistent with the evaluation of coverage over time and time to discovery. It is therefore possible that the proposed seed prioritization and power scheduling work against one another. In order to draw any proper conclusion from the results on energy assignment deviations, another study should be conducted that more properly isolates what is being tested.

SpareFuzz was based on the intuition that fuzzer heuristics that better correspond to reality would yield better results than more arbitrary heuristics. SpareFuzz use of probabilities is a step away from the arbitrary heuristics and towards heuristics that more closely models reality. However, it appears that this change in design does not necessarily improve the results, as made evident by the results. As with all heuristics, both the new approach and the old approach compared in this study are approximations that aim to provide a good solution. A solution that does not require any heuristic at all, but is instead made possible by taking into consideration all data at hand would be the very best one. How far a heuristic is from this optimal solution is perhaps not as apparent as initially thought. Just because a heuristic might seem to be closer to reality does not necessarily mean that it will perform better.

With the limitations in the current evaluation, it is difficult to conclude whether probabilities are better or worse in all situations or only in some. By improving how probability data is estimated and comparing how the different design proposals work in isolation, a more thorough conclusion can be drawn. It is the author’s recommendation to compare the inclusion of probabilities within grey-box fuzzing more thoroughly before deciding on whether it is a promising direction within the field or not. This thesis gives an overview of what can be expected from probability-based grey-box fuzzing and what difficulties lie in its implementation, with the hope of inspiring future work in this area.
Bibliography


