Technische Universität München

Lehrstuhl für Sicherheit in der Informatik
Fakultät für Informatik

Fuzzing with Stochastic Feedback Processes

Konstantin Böttinger

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. Jens Großklags
Prüfer der Dissertation:	1.	Prof. Dr. Claudia Eckert
	2.	Prof. Dr. Felix Freiling

Die Dissertation wurde am 17.01.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.05.2019 angenommen.

Abstract

Motivated by the urgent need for secure software we construct new testing methods to improve current development lifecycles. We connect probability theory with current testing technologies by formulating feedback-driven fuzzing in the language of stochastic processes. This mathematical model allows us to translate deep results from probability theory into algorithms for software testing.

Our mathematical model captures fuzzing as a Makrov decision process. Translating processes with suitable characteristics yield testing algorithms with predefined behavior. We further enhance this stochastic approach with exact computation based on symbolic execution to reach deep layers of the targeted programs. Exploring the full capabilities of our model leads us to the application of reinforcement learning methods, which turns out to be a fruitful new direction in software testing.

Zusammenfassung

Motiviert durch den hohen Bedarf an sicherer Software werden neue Testmethoden bereitgestellt, um moderne Entwicklungsprozesse zu verbessern. Durch die Formulierung Feedback-basierten Fuzzings in der Sprache stochastischer Prozesse wird das Gebiet der Wahrscheinlichkeitstheorie mit gängigen Testtechnologien verbunden. Dieses mathematische Modell ermöglicht es, tiefgreifende Resultate aus der Wahrscheinlichkeitstheorie in Algorithmen zum Testen von Software zu übersetzen.

Das mathematische Modell erfasst Fuzzing als Markov Entscheidungsprozess. Das Übersetzen von Prozessen mit geeigneten Eigenschaften ergibt Testalgorithmen mit vordefiniertem Verhalten. Dieser stochastische Ansatz wird mit der Möglichkeit zur exakten Berechnung basierend auf symbolischer Programmausführung erweitert, um tiefe Ausführungsschichten des getesteten Programmes zu erreichen. Die Analyse aller Eigenschaften des aufgestellten mathematischen Modells führt schließlich zur Anwendung von Methoden des bestärkenden Lernens. Die vorgestellte Herangehensweise erweist sich als vielversprechende neue Richtung im Bereich des Softwaretestens.

Contents

1. Introduction **9**
- 1.1. Research Challenge . . . 9
- 1.2. Research Contribution . . . 10
- 1.3. Impact . . . 11

I. The Stochastic Process of Fuzzing **13**

2. Fuzzing Essentials **17**
- 2.1. Fuzzing Origins . . . 17
- 2.2. Modern Fuzzing . . . 18
  - 2.2.1. Bug Observation and Identification . . . 18
  - 2.2.2. Fuzzer Taxonomy . . . 20
- 2.3. Generic Architecture and Processes . . . 22

3. Markov Decision Processes **23**
- 3.1. Policies and Behavior . . . 23
- 3.2. Value Functions . . . 26

4. Fuzzing as a Markov Decision Process **29**
- 4.1. States . . . 29
- 4.2. Actions . . . 31
- 4.3. Rewards . . . 32

II. Fuzzing with Predefined Behavior **33**

5. Hunting Bugs with Lévy Flight Foraging **37**
- 5.1. Motivation . . . 38
- 5.2. Related work . . . 39
- 5.3. Lévy Flights in Input Space . . . 40
  - 5.3.1. Lévy Flights . . . 40
  - 5.3.2. Input Space Flights . . . 41
- 5.4. Quality Evaluation of Test Cases . . . 42
- 5.5. Fuzzing Algorithm . . . 43
- 5.6. Lévy Flight Swarms . . . 47
- 5.7. Implementation . . . 51
- 5.8. Discussion . . . 51

5.9. Conclusion . . . 52

6. Triggering Vulnerabilities Deeply Hidden in Binaries 55
6.1. Motivation . . . 56
6.2. Related Work . . . 58
6.3. The DeepFuzz Algorithm . . . 60
6.3.1. Initial Seed Generation . . . 60
6.3.2. Concolic Execution . . . 61
6.3.3. Distribution of Path Probabilities . . . 61
6.3.4. Path Selection . . . 62
6.3.5. Constrained Fuzzing . . . 64
6.3.6. Joining the Pieces . . . 65
6.4. Implementation and Evaluation . . . 66
6.4.1. Time and Memory Complexity . . . 67
6.5. Discussion . . . 68
6.6. Conclusion . . . 69

7. Guiding a Colony of Fuzzers with Chemotaxis 71
7.1. Motivation . . . 71
7.2. Guided Fuzzing . . . 73
7.2.1. Attractant Trace Generation . . . 73
7.2.2. Positive Chemotaxis . . . 74
7.2.3. Guided Fuzzing Algorithm . . . 75
7.2.4. Explorer Hierarchies . . . 77
7.2.5. Choices for f and g . . . 78
7.3. Implementation and Evaluation . . . 79
7.4. Discussion . . . 81
7.5. Conclusion . . . 82

III. Fuzzing with Learning Behavior 83

8. Reinforcement Fuzzing 87
8.1. Motivation . . . 88
8.2. Related Work . . . 88
8.3. Q-Learning . . . 89
8.4. Reinforcement Fuzzing Algorithm . . . 90
8.4.1. Initialization . . . 91
8.4.2. State Extraction . . . 91
8.4.3. Action Selection . . . 91
8.4.4. Mutation . . . 91
8.4.5. Reward Evaluation . . . 92
8.4.6. Q-Update . . . 92
8.4.7. Joining the Pieces . . . 92

8.5. Implementation and Evaluation 93
8.5.1. Target 93
8.5.2. Implementation 94
8.5.3. Evaluation 95
8.6. Discussion 100
8.7. Conclusion 100

9. Conclusion **101**
9.1. Markov Decision Machines 102
9.2. Outlook 102
9.2.1. Hierarchies of Learning Agents 102
9.2.2. Alternative Models 103

List of Figures

1.1. The stochastic process of fuzzing. 15

2.1. Generic architecture for feedback driven fuzzing. 22

3.1. Markov decision process. 23

4.1. Modeling fuzzing as a Markov decision process. 29

5.1. Fuzzing with Lévy flights. 37
5.2. Individual fuzzing algorithm. After initial seed generation the fuzzer enters the loop of test case generation, quality evaluation, adaptation of diffusivity, and test case update. 47
5.3. Swarm fuzzing algorithm. The swarm of fuzzers enters the loop of individual fuzzing, clustering with k-means, and relocation of individuals to positions of highest test case quality within respective clusters. 50

6.1. Fuzzing with symbolic reasoning. 55
6.2. Execution paths c'_i $(i = 1, ..., 4)$ for the initial seed X_0. 60
6.3. Execution paths c_{ij} for $i = 1, ..., n'$, $j = 1, ..., b_{max}$) and selected set $C_{high} = \{c_{11}, c_{22}, c_{33}, c_{44}\}$. 63
6.4. DeepFuzz main algorithm with parameters m, k_{min}, T_0, and b_{max}. 66
6.5. Evaluation of time and memory complexity. 68

7.1. Algorithm for guided fuzzing with input functions f, g and parameters n_E, n_W, and t'. 76
7.2. Attraction of a single explorer ($n_E = 1$) within the first 100 iterations, resulting in a decrease of δ from averaged 400 kbit down to 330 bit, where mutation ratio for the measured black-box fuzzer is $r = 4{*}10^{-5}$. Increasing d_2 from 4 to 14 causes a significant stronger attraction. 80
7.3. Attraction for $n_E = 5$ within the first 100 iterations and increasing $d_2 = 4, ..., 14$, causing a decrease of δ from averaged 400 kbit down to 190 kbit, where mutation ratio is $r = 8 * 10^{-4}$. 80

8.1. Reinforcement Fuzzing. 87
8.2. Reinforcement fuzzing algorithm. 93
8.3. $R_2 - n_g$ diagram for $w = 32$, $\epsilon = 0.1$, and $\gamma = 0.2$ for $Q-$learning (cyan) and baseline (blue) rewards. 97
8.4. $R_1 - n_g$ diagram for $w = 32$, $\epsilon = 0.1$, and $\gamma = 0.2$ for $Q-$learning (cyan) and baseline (blue) rewards. 97

8.5. $E_2 - n_g - A$ diagram for $w = 32$, $\epsilon = 0.1$, and $\gamma = 0.2$ for $|A| = 8$. Figure (a) shows an overall initial decrease of the Q function depending on the number n_g of generations, followed by explorative behavior (b). 99

9.1. Overview. 101

1. Introduction

We dive into this work with some introductory thoughts about our main research questions. By outlining our contributions from a bird's eye view we equip the reader with high-level directions that will help to keep track during the challenges that lie ahead. At the beginning of each of the following chapters we return to this bird's eye perspective to regain orientation. This guide through a detailed landscape of algorithms will eventually ready the reader for advanced fuzzing. At the very end of this work we connect our discoveries to one single map of the world. Embedding this map into a global atlas will illuminate our journey in an excitingly unknown context. We wish the reader to find inspiration and fruitful thoughts while exploring the world of fuzzing.

1.1. Research Challenge

The ever increasing complexity of software systems in the core infrastructures of society demands advanced methods for testing their robustness. In recent years we observe an increasing proliferation of serious software vulnerabilities in the technologies that surround us. The perfectly secure piece of software is far out of reach and common practice in hardening software often boils down to finding vulnerabilities before the adversary does. Undisclosed security-critical bugs known as *zero-days* will continue to emerge on the surface of black markets to attract players of a variety of backgrounds. A common strategy to decrease the risk of being successfully attacked is to increase the effort it takes to compromise our assets. From the perspective of practical risk assessment, the work at hand presents advanced methods to lower this probability in efficient ways. Efforts in reducing the attack surface and increasing attack efforts directly point us to research secure software development lifecycles.

Besides secure design and implementation, state-of-the-art in secure software engineering always includes several verification steps prior to release. In practice there is always a certain mismatch between functionality intended by the architect and actually provided by the implementation at hand. This mismatch gives rise to unintended and unexpected behavior in terms of security critical vulnerabilities. To discover such flaws a magnitude of different verification methods have emerged over time. From a practical point of view we especially need automated methods that allow us to systematically perform vulnerability analysis of software. The modern world of software engineering strongly requires fully automated testing tools that scale to the ever growing application landscape.

The nowadays most effective way to proceed in this direction is random testing of software, also called fuzzing. There exists a substantial diversity of test case generation strategies for random testing of software. All these approaches have in common to a greater or lesser extent the random generation of test cases with the aim of driving the

targeted program to an unexpected and possibly exploitable state. The prime advantage of fuzzing is its relative ease of use. Most software that processes any input data is a suitable target for random test generation and simple fuzzers are implemented in a short time. This ease of use comes with a lack of completeness: Fuzzing does not guarantee the absence of vulnerabilities but only reduces the probability of their existence. However, from point of view of practical risk assessment, decreasing the number of security critical vulnerabilities exactly corresponds to the required risk reduction.

Looking at state-of-the art in *random* testing, we see the discipline of randomness very much underrepresented. Rooted in ancient times, the theory of probability gained momentum in the correspondence of Gerolamo Cardano, Pierre de Fermat, and Blaise Pascal beginning in 1654 [1]. From Christiaan Huygens' 1657 discourse *De ratiociniis in ludo aleae* ("On Reasoning in Games of Chance") over Andrey Kolmogorov's foundations of the field in 1933 [2] up to the powerful methods of modern stochastics [3], the theory of randomness has developed into an influential and rich mathematical discipline. When comparing the deep results of this theory to simple random bit flips in state-of-the-art random testing, two major research questions arise:

- How can we connect probability theory to state-of-the-art software testing?
- How can we transfer the deep results from probability theory into the world of software testing in order to discover new algorithms?

This thesis answers both questions in a mathematically rigorous way.

1.2. Research Contribution

To answer the first question, we construct a mathematical model of fuzzing. This provides a common language that functions as gateway between both fields of research. To answer the second question, we identify stochastic structures and processes underlying this model and translate their essence into algorithms for software testing.

In Part I of this work we connect fuzzing with the field of stochastic processes. On the one hand, fuzzing in its essence deals with controlling a feedback loop in the sense of classic cybernetics [4]: The Fuzzer generates an input, injects it to the program under test, observes what the program does, and adapts its behavior for generating the next input accordingly. On the other hand, the rich field of probability theory offers deep results for feedback-driven stochastic processes. Formulating fuzzing in the language of mathematics enables us to directly transfer results from probability theory to fuzzing. Well established methods and search strategies proven to be stable and effective suddenly give rise to novel fuzzing algorithms. This way we open the door to a variety of new perspectives on software testing.

To guide our choice of perspectives we find inspiration in the field of biology. In Part II of this work we investigate fuzzing strategies inspired by animal foraging and swarm theory. We construct self-adaptive feedback loops for fuzzing and evaluate their efficiency on realistic targets. Further, we enhance the random nature of input generation with

deterministic and precise methods: The combination of fuzzing with symbolic reasoning turns out to be effective in reaching deep layers of the program under test. Part II deals with predefined behavior in the sense that input mutation and synchronization follows a fixed sequence of actions.

We research fuzzers with learning behavior in Part III. At a certain level of abstraction controlling the fuzzing loop can be interpreted as a game against the program. Motivated by the success in Backgammon [5, 6], Atari games [7], and the game of Go [8] we apply machine learning to fuzzing. Again, the mathematical model of Part I provides a direct interface to reinforcement learning. And in fact, the deep Q learning algorithm that achieved super-human behavior in [7] and [8] turns out to be an exciting new direction in software testing.

We draw the bird's eye perspectives in simple figures at the beginning of each part. Each such figure captures the essence of the respective contribution.

1.3. Impact

In the course of writing this thesis sixteen scientific publications [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24] emerged. These are just snapshots of an odyssey through IT security that eventually led to the core theme of the work at hand.

This thesis embeds ideas from papers presented at the IEEE Symposium on Security and Privacy [16, 20, 23], at the Conference on Detection of Intrusions and Malware and Vulnerability Assessment [17], in two journal papers [18, 21], and a book chapter [22].

Beyond pure research the presented ideas found resonance in the software industry. During a three-monthly stay at Microsoft Research Redmond in the summer of 2017 the approach of reinforcement fuzzing turned out to be a promising new approach in software testing. We expect that further efforts in the spirit of this thesis will soon impact security development lifecycles to yield value and good for society.

Part I.

The Stochastic Process of Fuzzing

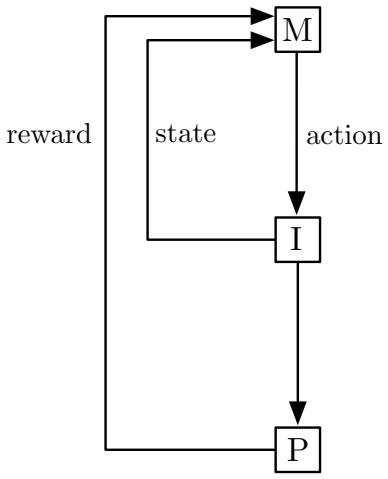


Figure 1.1.: The stochastic process of fuzzing.

In this part we connect fuzzing with stochastics. We formulate fuzzing in terms of stochastic processes, which allows us to construct a mathematical model of generic fuzzing architectures. The essence of this part is captured in Figure 1.1: The mutator M, input I, and program P, which are concepts of fuzzing, are connected via rewards, states, and actions, which in turn are concepts of certain stochastic processes. The mutaror M generates an input I that is injected into the program under test P. This input generation is interpreted as an action that causes a state transition and a reward. At the end of this part the reader will understand each aspect of this view.

First, we present the essential background necessary to understand state-of-the-art fuzzing in Chapter 2. This includes historical notes, an introduction of common terminology, and a short note on testing taxonomies as well as an abstraction of a generic architecture.

Second, we introduce the language required for mathematically modeling fuzzing in Chapter 3. We keep the discussion of Markov decision processes at an assessable level to keep the overall presentation as clear as possible.

Third, in Chapter 4 we capture the generic architecture abstracted in Chapter 2 in the language introduced in Chapter 3. This mathematical model of fuzzing provides the basis for Parts II and III of this work.

2. Fuzzing Essentials

In this chapter we introduce everything necessary to understand state-of-the-art fuzzing. We trace the development of random test generation beginning at its origins in the 1950s to the advanced feedback-driven frameworks for modern software testing. This leads us to characteristic properties of modern fuzzers, based on which we construct a generic architecture to obtain an abstract view on the fuzzing process. Extracting the essentials of fuzzing this way will ready us for the challenges of a formal analysis that lie ahead.

2.1. Fuzzing Origins

Reasoning about the correctness of the computer is as old as computing itself. Even if we go back to the stepped reckoner, a digital mechanical calculator invented by Leibniz [25], we find the abyss that ever since opened up between intent and reality of computation: Beyond calculation errors in the precision gearwork that drove fine mechanics technology in those days over its limits, a design error in the carrying mechanism was detected in 1893 - 199 years after construction of the machine. The first attempt to systematically reason about the correctness of a program can be dated back to 1949 when Turing indicated a general proof method for program correctness on three foolscap pages of text [26, 27]. Research of the following decades established the field of program verification [28, 29, 30], which aims for proving correctness of a program with respect to formal specifications and properties. However, proving correct system behavior for large programs suffers from explosion of possible states inherent in complex software. Further, verification techniques require a system model to prove properties of the program. Constructing such models counters the aim of fully automated testing and may even miss system properties [31].

Parallel to the discipline of formal verification another kind of software testing was established. As Gerald M. Weinberg [32] recalls: *"We didn't call it fuzzing back in the 1950s, but it was our standard practice to test programs by inputting decks of punch cards taken from the trash. We also used decks of random number punch cards. We weren't networked in those days, so we weren't much worried about security, but our random trash decks often turned up undesirable behavior."* Networking evolved and the rise of the internet exacerbated the situation of complex systems facing a lack of scalable testing methods. Software ever increased in size and complexity - so did the proliferation of software bugs - and finding flaws in systems becomes even harder when they are distributed. A computer connected to the ARPANET [33, 34] was much more likely to process data from untrusted sources. From security perspective, a system without air-gap means an increased attack surface, which was famously demonstrated by the Morris worm in 1988 [35]. Although testing with random program inputs was considered far inferior compared to the theory of formal verification (and sometimes even viewed as the *"worst*

case of program testing" [36]), it was applied as a cost-effective alternative in practical software engineering. Duran and Ntafos [37, 36] justified the use of random testing in the early 1980s by evaluating its effectiveness in relation to more formal methods. Random testing, meanwhile referred to as *monkey testing* due to the eponymous tool *The Monkey* released 1983 by Steve Capps to test user interfaces for the Macinthosh, was established among the programming practitioners of the late 1980s. But it was lacking the theoretical background needed to increase trust in this method. Just in the same fall of the year 1988 when the Morris worm spread, the perils of interconnected systems themselves gave rise to a systematic approach in the spirit of the 1950s random decks: When Barton Miller remotely connected to his Unix system during a fall thunderstorm, the rain caused noise on the line and thereby in the commands he entered to the shell leading programs to crash [38]. This motivated him to systematically execute programs with random and unstructured input data, which he referred to as *fuzzing* [39]. Since then the fuzzing discipline has evolved to an active area of research providing a rich diversity of fuzzing tools available, each focusing on specialized approaches. Fuzzing is nowadays the prevalent method used for detecting vulnerabilities in binaries. In a nutshell, inputs are randomly generated and injected into the target program with the aim to drive the program to an unexpected and exploitable state.

2.2. Modern Fuzzing

The overall goal of fuzzing a target executable is to drive it to an unexpected and unintended state. Informally, we want to cause error signals, program crashes, and timeouts and refer to such program behavior as a bug. In this section we first show how to actually sense bugs in a program during testing and discuss aspects of bug classification and criticality. Subsequently, we motivate currently applied search strategies that aim for maximization of code coverage. Such coverage information can be interpreted as a feedback mechanism that gives rise to feedback driven fuzzing.

2.2.1. Bug Observation and Identification

Finding bugs in the program under test requires a mechanism to sense them. Current state-of-the-art fuzzers detect bugs that cause the target to timeout or crash. In the following we also refer to such behavior as a vulnerability of the program.

Timeouts The former refers to delay or complete absence of an expected program response. In most situations such timeouts simply require a specified amount of time the fuzzer waits for a responds. Therefore, sensing timeouts is straight forward in most cases and current state-of-the-art fuzzers come with default time values around one second. Only very rare settings require more caution: The Windows Operating systems (from Windows Vista upwards) for example come with the *Timeout Detection and Recovery* functionality, which detects problems in the response from graphic cards and upon detection of a frozen GPU resets the relevant drivers. In such cases, where timeouts of

sub-systems of the fuzzing target are handled by the target itself or its execution environment, timeout handling of the target must be disabled or more advanced sensing mechanisms are required.

Crashes A program crash refers to termination of the program due to a failure condition. Such conditions can happen inside the processor or in processor-external hardware modules. The latter indicate failure conditions by sending an interrupt to the processor. Since such external interrupt signals are usually asynchronous to the processor clock, they are referred to as *asynchronous events*. Processor-internal failure conditions in turn are generated synchronous to the processor clock and we refer to them as *exception events*.

Our definition of interrupts and exceptions is compliant with standard texts on processor design [40] and the Intel x86 and x86-64 software developer manuals [41]. However, this distinction is not always consistently followed in the literature and even standard references on the Linux kernel [42] occasionally refer to interrupts as both, synchronous and asynchronous events. Considering the broad spectrum of different processor architectures, such vagueness of notation in the related literature seems natural: The ARM processor manuals, for example, include software interrupts (not to be confused with the hardware interrupts in our definition) as exceptions. Further, it depends on the processor architecture if the failure condition is labelled an interrupt or an exception. For example, a processor-external memory management unit detecting an unauthorized memory access indicates an interrupt, whereas an memory management unit integrated in the processor (as implemented most often in modern CPU designs) per our definition rises an exception. In any case, our definition of crashes is sufficient for the presentation of this thesis as it covers both, synchronously and asynchronously generated events: We are interested in software bugs and mainly abstract away the specific processor architecture.

The causes of crashes are manifold and typically fall into one of the following classes: Division error, invalid opcode, overflow, page fault, unauthorized memory access, and unauthorized call of a routine with higher privileges. The exact types of failure conditions depend on the specific processor. For example, the Intel x86 and x86-64 architecture defines exceptions and interrupts related to coprocessor segment overrun, floating-point errors, virtualization exceptions and many more (see [41] Chapter 6). Each time a crash occurs, the operating system takes care of its handling. For example, the Linux routine for processors implementing the x86 and x86-64 instruction set architecture proceeds as follows. Upon receiving such interrupt or exception, the processor stops execution of the current process, saves all process registers, and switches to the operating system event handler indicated by the interrupt descriptor table. The operating system handler in turn sends a signal to the target process that evoked the interrupt. To eventually sense the crash at software side, we catch the signals that are sent from the operating system to the target process and filter the fatal ones.

Each operating system comes with its own types of signals. For Unix-like operating systems, famous fatal signals include SIGABRT (abnormal termination signal), SIGSEGV (invalid memory access signal), SIGSYS (bad argument to system call), SIGFPE (erroneous arithmetic operation), and SIGILL (illegal instruction). We refer to the POSIX

programmers guide [43] for a complete list of fatal signals in the UNIX environment.

Software vulnerabilities come in a large spectrum of different characteristics, which motivated a diversity of research efforts to categorize them. For example, we could just take the criticality of bugs depending on their effect on the defined assets into account. If the asset to safeguard is availability of a server, a bug B_1 that crashes the server should be considered critical. If a second bug B_2 only crashes a server submodule that gets restarted automatically it does not affect the overall stability of the program and is less critical with regard to server availability. In contrast, if the asset is data confidentiality and B_2 allows an attacker to read out confidential data, it should be considered critical. The famous *Heartbleed* bug from 2014 that allowed remote read of protected memory in estimated up to 55% of popular HTTPS internet sites [44] belongs to the latter kind. Besides criticality there is a magnitude of other characteristics that give rise to a variety of different vulnerability taxonomies. However, discussing this active area of research is out of scope of this work and we refer to [45, 46, 47, 48] for a first overview. Practically, the *Common Vulnerabilities and Exposures* (CVE) data provides a standardized corpus of specific software bugs that facilitates identification and communication of concrete vulnerabilities. Similarly, the *Common Weakness Enumeration* (CWE) provides a more general accumulation of common software vulnerabilities separate from specific products.

Intuitively, the likelihood of finding a bug rises with the percentage of code that we execute. In fact, the idea that code coverage leads to bug coverage was proven fruitful in the early days of fuzzing [49]. As a natural evolution, coverage levels were not only reported as minimal adequacy criteria in development lifecycles, but also used as reward feedback during the actual fuzzing process to generate inputs that potentially explore new code regions. We systematically define such rewards in Section 4.3.

2.2.2. Fuzzer Taxonomy

In this section we briefly discuss a common basic taxonomy to classify modern fuzzing frameworks. First, we distinct fuzzers depending on the level of target information they have access to: While white-box fuzzers [50] have full sight on the target source code and therefore can theoretically gain detailed information about the program, black-box fuzzers [38, 51] are basically blind in the sense that they only sense program crashes or timeouts during testing. Grey-box methods are settled in between and often make use of instrumentation frameworks (such as Pin [52], Valgrind [53], DynamoRIO [54], Dyninst [55], DTrace [56], QEMU [57], and the like) to gain detailed information regarding program execution. Evolutionary and white-box fuzzers such as AFL, Driller (enhancing AFL with symbolic execution), EFS, Sage, Choronzon, Honggfuzz, libFuzzer, Kasan, Kcov, and BFF belong to this category. While binary instrumentation provides advanced test case generation based on runtime feedback, it comes with relatively high overhead (see [52] for a benchmark) and resulting moderate test case throughput. In contrast, black-box fuzzers (such as zzuf, Peach, and Radamsa) pitch test cases into the targeted binary without gathering feedback from dynamic instrumentation, which makes them significantly faster compared to feedback-driven fuzzers.

Second, we can distinct fuzzers with respect to the information they have regarding the

input format. Generation fuzzers create and mutate inputs with respect to such input structure information, which may come as a predefined or dynamically learned grammar but also a less formal format specification. In contrast, mutation fuzzers are unaware of the input format. Both classes have advanced representatives within modern state-of-the art fuzzers: Peach and SPIKE for example are generation fuzzers that deploy a grammar, while AFL, zzuf, VUzzer [58] and Radamsa are powerful examples of mutation fuzzers.

Third, we distinct fuzzers between host and network based fuzzers. In contrast to host based fuzzers, network frameworks have to keep a state machine to handle communication sequences over time.

It is easy to extend this basic taxonomy to much more distinction features, as discussed in [38] and [51]. However, even the three presented basic differentiators sometimes fail: In Part III of this work we present a fuzzer that learns a generalized grammar for input formats and therefore evolves from a pure mutation fuzzer towards a generation fuzzer over time.

2.3. Generic Architecture and Processes

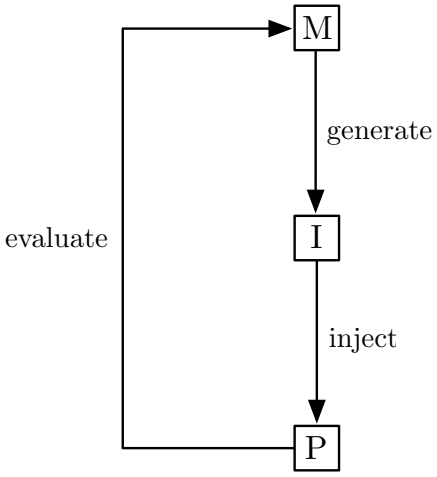


Figure 2.1.: Generic architecture for feedback driven fuzzing.

Now we are ready to abstract a generic architecture common for modern fuzzing frameworks and identify basic processes within the components. This abstract view will allow us model fuzzing in the language of mathematics and therefore acts as a bridge between stochastic analysis and software testing.

In principle, a fuzzer generates an input using a set of predefined actions for bit string manipulation and generation. It injects this input into the target under test and observes the result. While black-box fuzzers are limited to sensing crashes and timeouts, feedback driven frameworks gather detailed runtime information of the program executing the generated input, as discussed in Section 2.2.2. Subsequently, the fuzzer evaluates the extracted information from target execution and generates a new input based on this evaluation. This abstract loop is depicted in Figure 2.1.

3. Markov Decision Processes

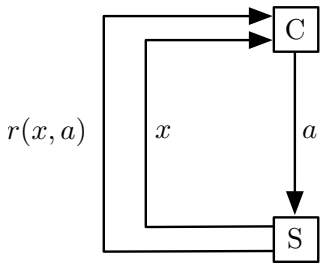


Figure 3.1.: Markov decision process.

In this section we introduce the formal background that gives us the expressiveness to formulate fuzzing in a mathematically rigorous shape. We abstract the concepts as introduced in Chapter 2 to obtain a generic model of fuzzing. This model directly connects fuzzing with the rich and deep theory of stochastic processes and allows us to infer fuzzing strategies based on mathematical reasoning. At the end of this chapter, the reader will fully understand the mathematical background of Markov decision processes as depicted in Figure 3.1.

3.1. Policies and Behavior

We begin the construction of our mathematical model of fuzzing with only two entities, namely a controller and a system. As depicted in Figure 3.1 we assume the controller C to interact with the system S via state observations x, actions a, and rewards $r(x, a)$. In this generic model the controller observes a state of the system and decides to take a corresponding action, which in turn results in a state transition of the system and an associated reward. Repeating this sequence gives rise to a feedback loop in which the controller aggregates rewards over time. The overall goal for the controller is to maximize the total reward. This scenario is commonly referred to as the reinforcement learning problem. Reinforcement learning as a subfield of machine learning is an active area of research and is best captured in the framework of Markov Decision Processes. The following introduction follows the notation of Szepesvári [59]. As usual in the probability theory literature, capital letters indicate stochastic variables while lower-cases denote their realizations.

Let $\mathcal{X}$ and $\mathcal{A}$ denote the set of all possible system states and controller actions, respectively. Let us first get an intuitive picture of a *probability kernel* $\mathcal{P}_0$. Assume the system

to be in state $x \in \mathcal{X}$. Upon observing x, the controller takes action $a \in \mathcal{A}$, which causes the system to perform a state transition associated with a corresponding reward to the controller. We assume the system to behave stochastically so that there is uncertainty regarding the new state $y \in \mathcal{X}$ of the system as well as the associated reward $\rho \in \mathbb{R}$. We can only provide the probability $\mathcal{P}_0(y, \rho|x, a)$ that the system transits from x to y with reward r upon action a. Formally, for each state $x \in \mathcal{X}$ and action $a \in \mathcal{A}$ let $\mathcal{P}_0(\,\cdot\,|x, a)$ denote a probability measure on the measurable space $(\mathcal{X}\times\mathbb{R}, \sigma(\mathcal{X}\times\mathbb{R}))$, where $\sigma(\mathcal{X}\times\mathbb{R})$ is the σ-algebra generated by $\mathcal{X} \times \mathbb{R}$. In other words, for each $(x, a) \in \mathcal{X} \times \mathcal{A}$ the probability kernel $\mathcal{P}_0$ gives rise to the probability space $\big(\mathcal{X} \times \mathbb{R}, \sigma(\mathcal{X} \times \mathbb{R}), \mathcal{P}_0(\,\cdot\,|x, a)\big)$. Then we define a *Markov Decision Process*

$$\mathcal{M} := (\mathcal{X}, \mathcal{A}, \mathcal{P}_0) \tag{3.1}$$

to be a set of states, actions, and an assigned probability kernel. As we will shortly see, $\mathcal{M}$ directly induces a stochastic process, which gives $\mathcal{M}$ its name. $\mathcal{P}_0$ directly determines the *state transition probability*

$$P(x, a, y) := \mathcal{P}_0\big(\{y\} \times \mathbb{R} \mid x, a\big) \tag{3.2}$$

for $(x, a, y) \in \mathcal{X}\times\mathcal{A}\times\mathcal{X}$, which denotes the probability that the system transits from state x to state y upon action a associated with any reward as indicated by the argument $\{y\}\times\mathbb{R}$. Further, $\mathcal{P}_0$ determines the expected reward upon action a and thus the *immediate reward function*

$$r:\ \mathcal{A} \to \mathbb{R} \tag{3.3}$$

$$r(x, a) := \mathbb{E}\left[R_{(x,a)}\right] \tag{3.4}$$

where the random variables $Y_{(x,a)}$ and $R_{(x,a)}$ are distributed according to

$$\big(Y_{(x,a)}, R_{(x,a)}\big) \sim \mathcal{P}_0(\,\cdot\,|x, a). \tag{3.5}$$

We assume all rewards to be bound such that

$$\exists \hat{R} > 0\ \forall (x, a) \in \mathcal{X} \times \mathcal{A} : |R_{(x,a)} \leq \hat{R}| \tag{3.6}$$

almost surely. This also bounds the expected reward

$$\|r\|_\infty = \sup_{(x,a)\in\mathcal{X}\times\mathcal{A}} |r(x, a)| \leq \hat{R}. \tag{3.7}$$

The repeated loop of state observation and action by the controller, state transition by the system, and resulting reward generation gives rise to the discrete time stochastic process $(X_t, A_t, R_{t+1})_{t\in\mathbb{N}}$, where transition states and associated rewards are distributed according to the probability kernel $(X_{t+1}, R_{t+1}) \sim \mathcal{P}_0(\,\cdot\,|X_t, A_t)$. The probability that the system transits from state $x \in \mathcal{X}$ to state $\in \mathcal{X}$ upon controller action $a \in \mathcal{A}$ is then given by the state transition probability

$$p(X_{t+1} = y|X_t = x, A_t = a) = P(x, a, y). \tag{3.8}$$

The reward of this transition is expected to be

$$\mathbb{E}[R_{t+1}|X_t, A_t] = r(X_t, A_t). \tag{3.9}$$

Next we formalize the process of action selection by the controller in more detail. We assume the controller makes decisions based an the whole history of actions, state transitions, and associated rewards. Formally, this is captured by an infinite sequence of probability kernels $(\pi_t)_{t\in\mathbb{N}}$ each mapping the process history to probability distributions over $\mathcal{A}$. The decision making by the controller is then determined by the current system state and experience of the past:

$$\forall a \in \mathcal{A} : \pi_t(a) = \pi_t(a|x_0, a_0, r_0, ..., x_{t-1}, a_{t-1}, r_{t-1}, x_t). \tag{3.10}$$

We refer to the sequence $(\pi_t)_{t\in\mathbb{N}}$ as a ***behavior*** and denote the set of all possible behaviors as Π. An initial system state $X_0 \in \mathcal{X}$ and a behavior fully govern the process $(X_t, A_t, R_{t+1})_{t\in\mathbb{N}}$. A controller that behaves according to $(\pi_t)_{t\in\mathbb{N}}$ accumulates the *total discounted sum of rewards*, also called *return*,

$$\mathcal{R} = \sum_{t=0}^{\infty} \gamma^t R_{t+1}, \tag{3.11}$$

where $\gamma \in [0, 1]$ is a discount factor. A lower value of γ prioritizes rewards in the near future while discounting rewards in the far future and vice versa. We already stated that the overall goal of the controller is to maximize its expected return. Such maximization requires the controller to behave optimally.

We can identify two special classes of behavior, namely *stochastic stationary policies* and *deterministic stationary policies*. Stochastic stationary policies

$$\pi : \mathcal{X} \to D(\mathcal{A}, \sigma(\mathcal{A})), \tag{3.12}$$

map system states to probability distributions (indicated by D) over the action space. For $\pi(X) = \pi' \in D(\mathcal{A}, \sigma(\mathcal{A}))$ we directly write

$$A_t \sim \pi(\,\cdot\,|X_t) \tag{3.13}$$

instead of $A_t \sim \pi'(\,\cdot\,|X_t)$ in the following. For our purposes this short notation does not introduce ambiguity. Deterministic stationary policies

$$\pi : \mathcal{X} \to \mathcal{A},\ A_t = \pi(X_t) \tag{3.14}$$

assign a fixed predefined action A_t to each observed state X_t. Such deterministic policies are special cases of stochastic stationary policies: Determinism corresponds to distributions with $\pi(A_t|X_t) = 1$ such that the probability mass of π for other actions than A_t is distributed only on a null set in $\mathcal{A} \setminus \{A_t\}$.

3.2. Value Functions

We define the *value function* for states $x \in \mathcal{X}$ to be the expected total reward

$$V^{\pi}(x) := \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | X_0 = x\right] \tag{3.15}$$

the controller accumulates when behaving according to $\pi \in \Pi$. This gives rise to the *optimal value function* $V^* : \mathcal{X} \to \mathbb{R}$ which indicates the highest possible return for the controller interacting with a system starting in state x. With this in mind we define the *optimal behavior* to achieve optimal return values for all initial states $x \in \mathcal{X}$. In other words, an optimal behavior yields the optimal return value

$$V^*(x) = \sup_{\pi \in \Pi} V^{\pi}(x). \tag{3.16}$$

Similarly, we define the *action-value function*

$$Q^{\pi} : \mathcal{X} \times \mathcal{A} \to \mathbb{R},\ Q^{\pi}(x,a) := \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | X_0 = x, A_0 = a\right] \tag{3.17}$$

to be the expected return when initially reacting with action $a \in \mathcal{A}$ to system state $x \in \mathcal{X}$ and then following behavior $\pi \in \Pi$. Analog to the above, let $Q^*(x,a) : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ denote the *optimal action-value function*. We refer to an action that maximizes $Q(x, \cdot\,)$ as a *greedy action* and to a policy that always prioritizes greedy actions as a *greedy policy*. With this notation in mind we can already state the Bellman equation

$$V^{\pi}(x) = r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} P(x, \pi(x), y) V^{\pi}(y). \tag{3.18}$$

With the Bellman operator

$$T^{\pi} : \mathbb{R}^{\mathcal{X}} \to \mathbb{R}^{\mathcal{X}},\ \left(T^{\pi} V\right)(x) := r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} P(x, \pi(x), y) V^{\pi}(y) \tag{3.19}$$

Equation (3.18) becomes

$$T^{\pi} V^{\pi} = V^{\pi}. \tag{3.20}$$

Analog, the Bellman optimality operator

$$T^* : \mathbb{R}^{\mathcal{X}} \to \mathbb{R}^{\mathcal{X}},\ \left(T^* V\right)(x) := \sup_{x \in \mathcal{A}} \{r(x,a) + \gamma \sum_{y \in \mathcal{X}} P(x,a,y) V(y)\} \tag{3.21}$$

gives rise to the Bellman optimality equation

$$T^* V^* = V^*. \tag{3.22}$$

With the operators

$$T^{\pi} : \mathbb{R}^{\mathcal{Z}\times\mathcal{A}} \to \mathbb{R}^{\mathcal{X}\times\mathcal{A}},\ (T^{\pi}Q)(x,a) := r(x,a) + \gamma \sum_{y\in\mathcal{X}} P(x,a,y)Q(y,\pi(y)) \tag{3.23}$$

and

$$T^{*} : \mathbb{R}^{\mathcal{X}\times\mathcal{A}} \to \mathbb{R}^{\mathcal{Z}\times\mathcal{A}},\ (T^{*}Q)(x,a) := r(x,a) + \gamma \sum_{y\in\mathcal{X}} P(x,a,y) \sup_{a'\in\mathcal{A}} Q(y,a') \tag{3.24}$$

this yiels similar equations

$$T^{\pi}Q^{\pi} = Q^{\pi} \text{ and} \tag{3.25}$$

$$T^{*}Q^{*} = Q^{*} \tag{3.26}$$

for the action-value functions. With $\gamma \in [0,1)$ the operators are contractions and the fixed-point theorem of Banach guarantees the existence and uniqueness of solutions, as disucussed in the functional analysis literature [60]. The Bellman equations shall for now close our presentation of Markov decision processes and we refer to Szepesvári [59] for a more comprehensive introduction to this theory. The Bellman equations will guide us in constructing reward maximization strategies in later chapters. In essence, they allow us to break down our overall goal of maximizing code coverage during fuzzing into smaller and local subproblems. This motivates the algorithms of Part II, where we predefine policies that lead to determined fuzzing behavior. Further, the Bellman equations give rise to the algorithms for reinforcement learning fuzzers as disussed in Part III. Such algorithms mimic learning behavior and self-adapt their policies as given in Equation (3.10).

4. Fuzzing as a Markov Decision Process

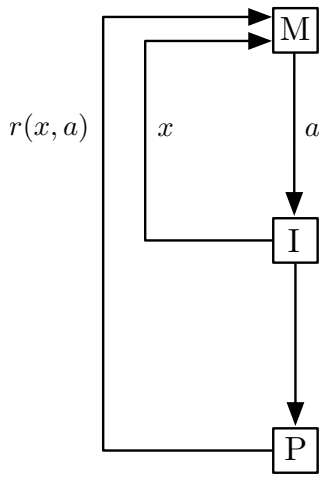


Figure 4.1.: Modeling fuzzing as a Markov decision process.

Now that we know the nature of fuzzing and the language of decision processes as formulated in Chapter 3 we enter the final phase of constructing our mathematical model. In the following we formulate each part of the generic architecture for feedback fuzzers (as presented in Section 2.3 and depicted in Figure 2.1) in the language of probability theory. This reformulation directly connects fuzzing with the theory of stochastic processes. Translating all essential aspects of fuzzing into abstract system states, controller actions, and process rewards opens the door for applying powerful methods from the field of search optimization and reinforcement learning. In the following sections we map the states x, actions a, and rewards $r(x, a)$ of a Markov decision process $\mathcal{M}$ (see Equation 3.1 on page 24) to their fuzzing counterparts as depicted in Figure 4.1.

4.1. States

In this section we introduce the states of feedback driven fuzzing as input strings of symbols. The input $x \in \Sigma^*$ as a string of symbols within an alphabet Σ further gives rise to a multitude of string features we could take into account for adapting the policy (as defined in Equations 3.12 and 3.14 on page 25). The spectrum reaches from processed characteristics such as entropy of fractions of the input to augmented input fractions obtained from dynamic taint analysis. While we operate on the bit level in Parts II and

III of this work, we do not limit our approach to this choice and formulate our model to be compatible with generic alphabets Σ.

States as Raw Input Strings

We can consider the system that the reinforcement learning agent learns to interact with to be a raw input string of alphabet symbols Σ. To realize this, we define the states that the agent observes to be substrings of consecutive symbols within such an input. Formally, let Σ denote a finite set of symbols. The set of possible target program inputs $\mathcal{I}$ written in this alphabet is then defined by the Kleene closure $\mathcal{I} := \Sigma^*$. For an input string $x = (x_1, ..., x_n) \in \mathcal{I}$ let

$$S(x) := \{(x_{1+i}, ..., x_{m+i}) \mid i \geq 0,\ m + i \leq n)\} \tag{4.1}$$

denote the set of all substrings of x. Clearly, $\cup_{x \in \mathcal{I}} S(x) = \mathcal{I}$ holds. We define the states of our Markov decision process to be

$$\mathcal{X} := \mathcal{I} = \Sigma^*. \tag{4.2}$$

In the following, $x \in \mathcal{I}$ denotes an input for the target program and $x' \in S(x) \subset \mathcal{I}$ a substring of this input.

State Features

Based on the raw input strings we can extract further refined state features as indicated in the following examples.

Offset and Width Again, let $\mathcal{I} = \Sigma^*$ denote the input space. For the given seed $x \in \mathcal{I}$ we can extract a strict substring $x' \in S(x)$ at offset $o \in \{0, ..., |x| - |x'|\}$ of width $|x'|$ as state features. In words, the reinforcement agent observes a fragment of the whole system via the substring x'. In this setting we can specifically define actions to move the offset and vary the width of the observed substring.

String Entropy Let $|x'| = n$ denote the length of a string $x' = (x'_1, ..., x'_n) \in \mathcal{I}$. The entropy of x' is then defined as

$$H(x') = \sum_{s \in \sigma(x')} p_s \log(p_s^{-1}), \tag{4.3}$$

where $\sigma(x')$ denotes the set of symbols represented in x' and p_s the probability of appearance of symbol s in string x'. As possible state we could take this entropy of substrings of an input x into account. The fuzzer then observes the changes in entropy during the fuzzing process.

Tainted Bytes We can further augment the input with taint information extracted from target program execution. Here, each symbol in x' is tracked with regard to subroutines that access it during execution. Symbols that are processed by the same subroutines are then grouped together and assigned with a label. These labels can then be taken into account as features for reinforcement learning. For example, Cui et al. [61] can automatically detect record sequences and types in the input by identification of chunks based on taint tracking input data in respective subroutine calls. Similarly, the authors of [62] apply dynamic tainting to identify failure-relevant inputs. Another recent approach was proposed by Höschele et al. [63], who mine input grammars from valid inputs based on feedback from dynamic instrumentation of the target by tracking input characters.

Since state-of-the-art taint tracking is computationally too expensive, we leave this set of features for future work.

4.2. Actions

In this section we introduce reinforcement fuzzing actions as rewriting rules for symbols in x.

Similar to the one-step rewriting relations in a semi-Thue system we define the set of possible actions $\mathcal{A}$ of our Markov decision process to be random variables mapping substrings of an input to probabilistic rewriting rules

$$\mathcal{A} := \{a : \mathcal{I} \to (\mathcal{I} \times \mathcal{I},\ \mathcal{F}, P) \mid a \sim \pi(x)\}\,, \tag{4.4}$$

where $\mathcal{F} = \sigma(\mathcal{I} \times \mathcal{I})$ denotes the σ-algebra of the sample space $(\mathcal{I} \times \mathcal{I})$ and P gives the probability for a given rewrite rule.

In the upcoming sections of this work we define both probabilistic and deterministic actions. This is still in line with our definition in Equation 4.4, where deterministic actions $a(x) = (x, x')$ correspond to $P((x, x')) = 1$ almost surely. Examples for probabilistic actions are random bit flips and shuffling bytes and sequences of bytes within x, while deterministic actions include string manipulation based on symbolic execution and insertion of dictionary tokens.

The choice of actions is a crucial design decision for feedback driven fuzzing. We experiment with a whole set of different actions in the following presentation and describe them in further detail in the corresponding sections. In fact, one major difference between Part II and III is how the fuzzer chooses between a set of given actions: While Part II discusses predefined behavior, where actions are given by a deterministic policy $a = \pi(x)$, Part III deals with actions $a \sim \pi(x)$ distributed according to a stochastic policy. In the latter case, we will show that reinforcement fuzzing is able to learn a high rewarding policy, i.e. picking high rewarding actions given observed states x.

4.3. Rewards

We define rewards for both characteristics of the performed action and program execution of the generated input independently, i.e.

$$R(x, a) = E(x) + G(a). \tag{4.5}$$

As described in Chapter 3, the stochastic variables $(y(x, a), R(x, a))$ are distributed according to $P_0(\cdot|x, a)$. G is provided by performing action a on x to generate a new mutation and E measured during execution of the target program with input x.

We experiment with E providing number of newly discovered basic blocks, execution path length, and the execution time of the target that processes the input x. Formally, let c_x denote the execution path the target program takes when processing input x and $B(c_x)$ the set of unique basic blocks of this path. Here, we define a basic block to be a sequence of instructions without branch instructions between block entry and exit. Given a history of previously processed inputs $I' \subset I$ we can write the number of newly discovered blocks as

$$E_1(x, I') := \left| B(c_x) \setminus \left(\bigcup_{\chi \in I'} B(c_\chi) \right) \right|. \tag{4.6}$$

Another choice of E is taking the execution time $E_2 = T(x)$ of the target into account. Similarly, we could define G to be the time it takes to generate a mutation based on the seed x. This would reinforce the fuzzer to find a balance between coverage advancements and action processing costs.

The idea to generate program inputs that maximize execution path coverage in order to trigger vulnerabilities has been discussed in the field of test case prioritization some time ago, see e.g. [64] and [65] for a comparison of coverage-based techniques. Rebert et al. [66] discuss and compare methods to gain optimal seed selection with respect to fuzzing and their findings support our decision to select code coverage for evaluating the quality of test cases.

We could further introduce rewards based on the execution graph. For example, it is conceivable to distribute negative rewards for execution paths that correspond to error handling code or in turn reward paths that enter a desired code area. We leave such types of reward for future work and concentrate on coverage and time in the following.

Part II.

Fuzzing with Predefined Behavior

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."

John von Neumann

In this second part of our presentation we discuss fuzzing with predefined behavior. Now that we have a mathematical model for fuzzing we can identify stochastic structures and processes underlying this model. The language of Markov decision processes allows us to directly translate the nature stochastic processes with well known behavior into effective algorithms for software testing.

As introduced in Equation 3.10 we refer to the sequence $(\pi_t)_{t\in\mathbb{N}}$ as a *behavior* and denote the set of all possible behaviors as Π. We can generally distinct the behaviors in two classes: Policies based on stochastic processes whose behavior is well researched, and policies based on learning processes whose behavior depends on the applied machine learning methods. In this second part we focus on the former class.

In Chapter 5 we investigate certain processes called Lévy flights that provably minimize search time in specific situations that suffice our conditions. The application of Lévy flights yields self-adaptive fuzzing behavior by adjusting the process parameters according to the feedback reward.

While the approach of Chapter 5 yields generally good results, it is purely stochastic in nature. However, reaching deep layers of a targeted program sometimes requires exact calculations, e.g. of checksums within the input. To pass the first parsing layers of the program we enhance our mathematical model based on stochastics with powerful formal methods based on symbolic execution. This upgrade provides the best characteristics of both worlds: The properties of search optimization from stochastic processes and partial input correctness required for deep fuzzing. We discuss this approach in detail in Chapter 6.

With a slight reinterpretation, the above given statement from John von Neumann holds: Upgrading purely stochastic fuzzing with symbolic execution comes with a high price. Such formal methods are computationally expensive and slow our algorithms down. Therefore, instead of packing our efficient stochastic fuzzers with formal sandbags, in Chapter 7 we give them hints and guide them towards high rewarding input regions. This introduces an elegant way to combine stochastic with formal methods while keeping the overall fuzzing process efficient. With our approach we actually can consider arithmetical methods of producing random digits without sacrificing efficiency.

5. Hunting Bugs with Lévy Flight Foraging

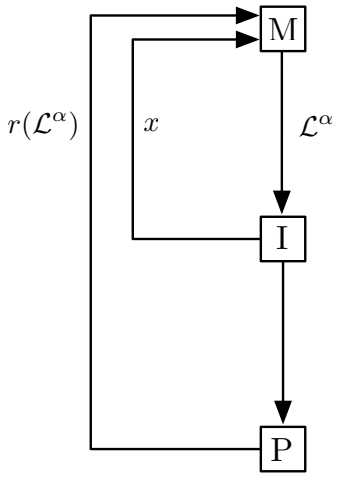


Figure 5.1.: Fuzzing with Lévy flights.

In this chapter we present a method for random testing of binary executables inspired by biology. In our approach we introduce the first fuzzer based on a mathematical model for optimal foraging. To minimize search time for possible vulnerabilities we generate test cases with Lévy flights in the input space. In order to dynamically adapt test generation behavior to actual path exploration performance we define a suitable measure for quality evaluation of test cases. This measure takes into account previously discovered code regions and allows us to construct a feedback mechanism. By controlling diffusivity of the test case generating Lévy processes with evaluation feedback from dynamic instrumentation we are able to define a fully self-adaptive fuzzing algorithm.

The overall approach of this chapter is depicted in Figure 5.1. The mutator engine M, input I, and target program P are connected via rewards, states, and actions as introduced in Chapter 4 of Part I of this work. Within the space of actions $\mathcal{A}$ as defined in Equation 4.4 on page 31 we focus on a special subset $\mathcal{L}^\alpha \subset \mathcal{A}$ based on Lévy flights. While the global behavior is determined by Lévy flights, the actual shapes of the flights are automatically adapted via parameters α, which in turn are adjusted according to the reward $r(\mathcal{L}^\alpha)$. Here, the reward $R(x, a)$ as introduced in Equation 4.5 on page 32 is inferred from effects of Lévy flight actions denoted by $r(\mathcal{L}^\alpha)$.

5.1. Motivation

In the course of researching new effective search strategies we find similar problems in biology, particularly in the field of optimal foraging. A variety of biological systems let us observe optimal strategies for finding energy sources by simultaneously avoiding predators. When we identify sources of food with possible vulnerabilities in binary executables and predators with the overhead of execution runtime, we are inspired to adapt mathematical models of optimal foraging to test case generation. This approach enables us to take stochastic models of optimal foraging as a basis for input mutation. In particular we rely on Lévy flights to search for bug triggering test cases in input space.

Before summarizing our contributions we first give some short background on optimal foraging and the Lévy flight hypothesis.

Optimal Foraging Observing biological systems has led to speculation that there might be simple laws of motion for animals searching for sources of energy in the face of predators. Regardless of whether we look at bumblebees[67], fish and hunting marine predators in the sea [68, 69], grey seals [70], spider monkeys [71], the flight search patterns of albatrosses [72], the wandering of reindeer [73], the reaction pathways of DNA-binding proteins [74], or the neutralisation of pathogens by white blood cells [75], we can discover emerging movement patterns all those examples have in common. Mathematical modelling such common patterns is an active field of research in biology and is more generally referred to as *movement ecology*. While the physics of foraging [76] provides us several possible models our choice is not guided by accuracy with respect to the biological process but by minimization of software bug search time. This leads us to the special class of stochastic processes called *Lévy flights* which we discuss in more detail in Section 5.3.

Lévy Flight Hypothesis Within the variety of models for optimal foraging Lévy flights have several characteristic properties that show promise for software testing. In particular, the Lévy flight hypothesis accentuates the most significant property of these kinds of stochastic processes for our purposes. It states that Lévy flights minimize search time when foraging sources of food that are sparsely and randomly distributed, resting, and refillable. These assumptions match to the properties of bugs in software (with the interpretation that *refillable* translates to the fact that software bugs stay until fix). In addition to the mathematical Lévy flight hypothesis, the Lévy flight *foraging* hypothesis in theoretical biology states that these processes actually model real foraging behavior in certain biological systems due to natural selection. The Lévy flight hypothesis constitutes the major connection link between optimal foraging theory and random software testing.

Swarm Behavior While moving patterns of foraging animals inspire us to define the behavior of a single fuzzer, we are further guided by biology when accumulating multipe fuzzer instances to a parallelized testing framework. Again we take a look at nature to

discover a whole branch of science that researches swarm behavior [77]. For example, the ants of a colony collectively find the shortest path to a food source. Based on simple rules for modeling natural swarm behavior we construct a *fuzzing swarm* that mimics colony clustering observed in biology. Our algorithm navigates the fuzzing swarm without a central control and provides self-organization of the fuzzers as they flexibly adapt to the binary structure under test.

In this Chapter we propose a novel method for random software testing based on the theory of optimal foraging. In summary, we make the following contributions:

- We introduce a novel fuzzing method based on Lévy flights in the input space in order to maximize coverage of execution paths.
- We define a suitable measure for quality evaluation of test cases in input space with respect to previously explored code regions.
- In order to control diffusivity of the test generation processes we define a feedback mechanism connecting current path exploration performance to the test generation module.
- We enable self-adaptive fuzzing behavior by adjusting the Lévy flight parameters according to feedback from dynamic instrumentation of the target executable.
- We aggregate multiple instances of such Lévy flights to fuzzing swarms which reveal flexible, robust, decentralized, and self-organized behavior.
- We implement the presented algorithm to show the feasibility of our approach.

The remainder of this Chapter is organized as follows. In Section 5.2 we discuss related work. In Section 5.3 we present necessary background on Lévy flights and show how to construct them in input space. We define a quality measure for generated test cases in Section 5.4, introduce our self-adapting algorithm for individual fuzzers in Section 5.5, and construct a swarm of multiple fuzzing instances in Section 5.6. Next, we give details regarding our implementation in Section 5.7 and discuss properties, possible modifications, and expansions of the proposed algorithm in Section 5.8. The chapter concludes with a short outlook in Section 5.9.

5.2. Related work

For definition of our quality measure for test cases we build upon code coverage heuristics as discussed in Section 4.3. The work of Cha et al. [78] is distantly related to a substep of our approach in the sense that they apply dynamic instrumentation to initially set the mutation ratio. However, they use completely different methods based on symbolic execution. Since symbolic preprocessing is very cost-intensive they further compute the mutation ratio only once per test, while our fuzzer presented in this chapter consistently self-adapts its mutation behavior during the whole fuzzing campaign.

Lévy flights have been studied extensively in mathematics and we refer to Zaburdaev et al. [79] and the references therein for a comprehensive introduction to this field. Very recently Chupeau et al. [80] connected Lévy flights to optimal search strategies and minimization of cover times.

5.3. Lévy Flights in Input Space

In this section we give the necessary background on Lévy flights and motivate their application. With this background we then define Lévy flights in input space.

5.3.1. Lévy Flights

Lévy flights are basically random walks in which step lengths exhibit power law tails. We aim for a short and illustrative presentation of the topic and refer to Zaburdaev et al. [79] for a comprehensive introduction. Pictorially if a particle moves stepwise in space while randomly choosing an arbitrary new direction after each step, it describes a Brownian motion. If in addition the step lengths of this particle vary after each step and are distributed according to a certain power law, it describes a Lévy flight.

Formally, Lévy processes comprise a special class of Markovian stochastic processes, i.e. collections of random variables

$$(L_t),\ t \in T \tag{5.1}$$

defined on a sample space Ω of a probability space $(\Omega, \mathcal{F}, P)$, mapping into a measurable space $(\Omega', \mathcal{F}')$, and indexed by a totally ordered set T. In our case Ω' refers to the discrete input space of the program and the index *time* T models the discrete iterations of test case generation, so we can assume $T = \mathbb{N}$. The process $(L_t)_{t \in T}$ is said to have *independent increments* if the differences

$$L_{t_2} - L_{t_1}, L_{t_3} - L_{t_2}, ..., L_{t_n} - L_{t_{n-1}} \tag{5.2}$$

are independent for all choices of $t_1 < t_2 < ... < t_n \in T$. The process $(L_t),\ t \in T$ is said to be *stationary*, if

$$\forall t_1, t_2 \in T, h > 0 :\ L_{t_1+h} - L_{t_1} \sim L_{t_2+h} - L_{t_2}, \tag{5.3}$$

i.e. increments for equal time intervals are equally distributed. A Lévy process is then formally defined to be a stochastic process having independent and stationary increments. The additional property

$$L_0 = 0\ a.s. \tag{5.4}$$

(i.e. almost surely) is sometimes included in the definition, but our proposed algorithm includes starting points other than the origin.

To construct a Lévy process $(L_n)_{n\in\mathbb{N}}$ we simply sum up independent and identically distributed random variables $(Z_n)_{n\in\mathbb{N}}$, i.e.

$$L_n := \sum_{i=1}^{n} Z_i. \tag{5.5}$$

The process $(L_n)_{n\in\mathbb{N}}$ is Markovian in the sense that

$$P(L_n = x_n | L_{n-1} = x_{n-1}, ..., L_0 = x_o) \tag{5.6}$$
$$= P(L_n = x_n | L_{n-1} = x_{n-1}), \tag{5.7}$$

which simplifies a practical implementation. If the distribution of step lengths in a Lévy process is heavy-tailed, i.e. if the probability is not exponentially bounded, we call the process a *Lévy flight*. Such processes generalize Brownian motion in that their flight lengths l are distributed according to the power law

$$p(l) \sim |l|^{-1-\alpha}, \tag{5.8}$$

where $0 < \alpha < 2$. They exhibit infinite variance

$$< l^2 >= \infty \tag{5.9}$$

which practically results in sometimes large jumps during search process. In fact, the ability to drive a particle very long distances within a single step gives Lévy flights their name. While Brownian motion is a suitable search strategy for densely distributed targets, Lévy flights are more efficient than Brownian motion in detecting widely scattered (software) bugs. Although there is much to say about the theoretical aspects of this class of stochastic processes we basically refer to the power law in equation (5.8) in the following. Smaller values of α yield a heavier tail (resulting in frequent long flights and super-diffusion) whereas higher values of α reveal a distribution with probability mass around zero (resulting in frequent small steps and sub-diffusion). In Section 5.5 we adapt α according to feedback information from dynamic instrumentation of the targeted binary.

As indicated in Section 5.1 Lévy flights are directly connected to the minimal time it takes to cover a given search domain. We refer to [80] for recent results regarding minimization of the mean search time for single targets.

5.3.2. Input Space Flights

Next we construct Lévy flights in the input space of binary executables under test. Therefore, assume the input to be a bit string of length N. If we simply wanted an optimal search through the input space without any boundary conditions, we would construct a one-dimensional Lévy flight in the linear space $\{0, ..., 2^N\}$. However, our aim is not input space coverage but execution code coverage of the binary under test. In this section we construct a stochastic process in input space with the properties we need for the main fuzzing algorithm presented in Section 5.5.

First, we divide the input into n segments of size $m = \frac{N}{n}$ (assuming without loss of generality that N is a multiple of n). We then define two Lévy processes, one in the space of offsets $\mathcal{O} = \{1, ..., n\}$ and one in the space of segment values $\mathcal{S} = \{1, ..., 2^m\}$. With underlying probability spaces $(\Omega_1, \mathcal{F}_1, P_1)$ and $(\Omega_2, \mathcal{F}_2, P_2)$ we define the one-dimensional Lévy flights

$$L_t^1 : \Omega_1 \to \mathcal{O} \tag{5.10}$$

$$L_t^2 : \Omega_2 \to \mathcal{S} \tag{5.11}$$

with index space $t \in \mathbb{N}$ and corresponding power law distribution of flight lengths l

$$p_j(l) \sim |l|^{-1-\alpha_i}, \ j = 1, 2 \tag{5.12}$$

where $0 < \alpha_i < 2$. While $(L_t^1)_{t\in\mathbb{N}}$ performs a Lévy flight in the offset parameter space, $(L_t^2)_{t\in\mathbb{N}}$ performs Lévy flights within the segment space indicated by the offset. Regarding the initial starting point (L_0^1, L_0^2) we assume a given seed input. We choose an arbitrary initial offset $L_0^1 \in \mathcal{O}$ and set the initial value of L_0^2 according to the segment value (with offset L_0^1) of the seed input.

By setting different values of α we can control the diffusivity of the stochastic processes $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$. If we find a combination of offset and segment values of high quality the fuzzer should automatically explore nearby test cases, which is realized by higher values of $0 < \alpha_i < 2$. Similarly if the currently explored region within input space reveals low quality test cases, the fuzzer should automatically adapt to widen its search pattern by decreasing α. Therefore, we first have to define a quality measure for test cases.

5.4. Quality Evaluation of Test Cases

In this section we define a quality measure for generated test cases. We aim for maximal possible code coverage in a finite amount of time, so we evaluate a single input by its ability to reach previously undiscovered execution paths. In other words, if we generate an input that drives the program under test to a new execution path, this input gets a high quality rating. Therefore we have to define a similarity measure for execution traces. We will then use this measure in Section 5.5 as feedback to dynamically adapt diffusivity of the test case generation process.

The field of test case prioritization provides effective methods for coverage-based rating (see [64] and [65] for a comparison). We adapt the method of prioritizing test cases by additional basic block coverage. As introduced in Section 5.3 we assume inputs for the program under test to be bit strings of size N and denote the space of all possible inputs as $\mathcal{I} = \{0, ..., 2^N\}$. Our challenge can then be formulated as follows. Given a subset of already generated input values $\mathcal{I}' \subset \mathcal{I}$, how do we measure the quality of a new input $x_0 \in \mathcal{I}$ with respect to maximal code coverage? For a given $x_0 \in \mathcal{I}$ let c_{x_0} denote the execution path the program takes for processing x_0. Intuitively we would assign a high quality rating to the new input x_0 if it drives the targeted program to a previously

undiscovered execution path, i.e. if c_{x_0} differs significantly from all previously explored execution paths $\{c_x | x \in \mathcal{I}'\}$. To measure this path difference we take the amount of newly discovered basic blocks into account. Here we refer to a *basic block* as a sequence of machine instructions without branch instructions between block entry and block exit. Let $B(c_x)$ denote the set of basic blocks of execution path c_x. The set of newly discovered basic blocks while processing a new test case x_0 given already executed test cases $\mathcal{I}' \subset \mathcal{I}$ is then

$$B(c_{x_0}) \setminus \left(\bigcup_{x \in \mathcal{I}'} B(c_x) \right). \tag{5.13}$$

We define the number $E(x_0, \mathcal{I}')$ of these newly discovered blocks as

$$E(x_0, \mathcal{I}') := \left| B(c_{x_0}) \setminus \left(\bigcup_{x \in \mathcal{I}'} B(c_x) \right) \right|, \tag{5.14}$$

where $|A|$ denotes the number of elements within a set A. The number $E(x_0, \mathcal{I}')$ indicates the number of newly discovered basic blocks when processing x_0 with respect to the already known basic blocks executed by the test cases within $\mathcal{I}'$. Intuitively $E(x_0, \mathcal{I}')$ gives us a quality measure for input x_0 in terms of maximization of basic block coverage. In order to construct a feedback mechanism we will use a slightly generalized version of this measure to control diffusivity of the input generating Lévy processes in our fuzzing algorithm in Section 5.5.

5.5. Fuzzing Algorithm

In this section we present the overall fuzzing algorithm. Our approach uses stochastic processes (i.e. Lévy flights as introduced in Section 5.3) in the input space to generate test cases. To steer the diffusivity of test case generation we provide feedback regarding the quality of test cases (as defined in Section 5.4) to the test generation process in order to yield self-adaptive fuzzing.

We first prepend an example regarding the interplay between input space coverage and execution path coverage to motivate our fuzzing algorithm. Consider a program which processes inputs from an input space $\mathcal{I}$. Our aim is to generate a subset $\mathcal{I}' \subset \mathcal{I}$ of test cases (in finite amount of time) that yields maximal possible execution path coverage when processed by the target program. Further assume the program to reveal deep execution paths (covering long sequences of basic blocks) only for 3% of the inputs $\mathcal{I}$, i.e. 97% of inputs are inappropriate test cases for fuzzing. Since we initially cannot predict which of the test cases reveals high quality (determined by e.g. the execution path length or the number of different executed basic blocks), one strategy to reach good code coverage would be black-box fuzzing, i.e. randomly generating test cases within $\mathcal{I}$ hoping that we eventually hit some of the 3% high quality inputs. We could realize such an optimal search through input space with highly diffusive stochastic processes, i.e. Lévy flights as presented in Section 5.3.

As mentioned above the Lévy flight hypotheses predicts an effective optimal search through input space due to their diffusivity properties. On the one hand this diffusivity guarantees us reaching the 3% with very high probability. On the other hand, once we have reached input regions within the 3% of high quality test cases, the same diffusivity also guarantees us that we will leave them very efficiently. This is why we need to adapt the diffusivity of the stochastic process according to the quality of the currently generated test cases. If the currently generated test cases reveal high path coverage, the Lévy flight should be localized in the sense that it reduces its diffusivity to explore nearby inputs. In turn, if the currently generated test cases reveal only little coverage, diffusivity should increase in order to widen the search for more suitable input regions. By instrumenting the binary under test and applying the quality evaluation of test cases introduced in Section 5.4 we are able to feedback coverage information of currently explored input regions to the test case generation algorithm. In the following we construct a self-adaptive fuzzing strategy that automatically expands its search when reaching low quality input regions and focuses exploration when having the feedback of good code coverage.

Initial Seed We start with an initial non-empty set of input seeds $X_0 \subset \mathcal{I}$. As described in Section 5.3 we assume the elements $x \in X_0$ to be bit strings of length N and divide each of them into n segments of size $m = \frac{N}{n}$ (assuming without loss of generality that N is a multiple of n). Practically the input seeds X_0 can be arbitrary files provided manually by the tester, they may not even be valid with regard to the input format of the program under test. We further set two initial diffusive parameters $0 < \alpha_1, \alpha_2 < 2$ and an initial offset $q_0 \in \{1, ..., n\}$.

Test Case Generation The test case generation step takes as input a test case x_0, diffusion parameters α_1 and α_2, an offset number $q_0 \in \{1, ..., n\}$, and a natural number $k_{gen} \in \mathbb{N}$ of maximal test cases to be generated. It outputs a set X_{gen} of k_{gen} new test cases $X_{gen} \in \mathcal{I}$.

As introduced in Section 5.3 we refer to the offset space as $\mathcal{O} = \{1, ..., n\}$ and to the segment space as $\mathcal{S} = \{1, ..., 2^m\}$. We denote with $x_0(q_0)$ the segment value of input x_0 at offset q_0. For the Lévy flights

$$L_t^1 : \Omega_1 \to \mathcal{O} \tag{5.15}$$

in the offsets $\mathcal{O}$ and

$$L_t^2 : \Omega_2 \to \mathcal{S} \tag{5.16}$$

in $\mathcal{S}$ with flight lengths l distributed according to the power law

$$p_j(l) \sim |l|^{-1-\alpha_j}, \ j = 1, 2 \tag{5.17}$$

we set the initial conditions

$$L_0^1 = q_0 \text{ and} \tag{5.18}$$

$$L_0^2 = x_0(q_0), \tag{5.19}$$

respectively. Let $R(x_0, q_0, s_0)$ denote the bit string generated by replacing the value $x_0(q_0)$ of bit string x_0 at offset q_0 by a new value s_0. Both stochastic processes $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ are then simulated for k_{gen} steps to generate the k_{gen} new test cases

$$x_1 := R\big(x_0, L_0^1, L_1^2\big) \tag{5.20}$$
$$x_2 := R\big(x_1, L_1^1, L_2^2\big) \tag{5.21}$$
$$\ldots$$
$$x_{t+1} := R\big(x_t, L_t^1, L_{t+1}^2\big) \tag{5.22}$$
$$\ldots$$
$$x_{k_{gen}} := R\big(x_{k_{gen}-1}, L_{k_{gen}-1}^1, L_{k_{gen}}^2\big). \tag{5.23}$$

For simplicity of notation in this definition we identify the values L_t^j with their respective binary representations (as bit string). In words, we start with the initial test case x_0 and replace its segment content at offset $L_0^1 = q_0$ with the new value L_1^2, which is the value in segment space $\mathcal{S} = \{1, ..., 2^m\}$ that we get when taking a first random step with the Lévy flight $(L_t^2)_{t\in\mathbb{N}}$. This yields x_1. We get the next test case x_2 by considering the just generated x_1, setting the offset according to $(L_t^2)_{t\in\mathbb{N}}$, and then replacing the content of the segment indicated by this offset by a new segment value chosen by $(L_t^2)_{t\in\mathbb{N}}$. We proceed with this algorithm until the set

$$X_{gen} := \{x_1, ..., x_{k_{gen}}\} \tag{5.24}$$

of k_{gen} new test cases is generated.

Quality Evaluation The quality evaluation step takes as input two sets of test cases $X_{gen}, \mathcal{I}' \subset \mathcal{I}$ and outputs a quality rating $\tilde{E}(X_{gen}, \mathcal{I}')$ of X_{gen} with respect to $\mathcal{I}'$. We already defined the number $E(x_0, \mathcal{I}')$ of newly discovered basic blocks for a single test case x_0 with respect to a given subset $\mathcal{I}' \subset \mathcal{I}$ in Equation (5.14). To generalize this definition to a quality rating $\tilde{E}(X_{gen}, \mathcal{I}')$ of a set of test cases X_{gen} (with respect to $\mathcal{I}'$) we define the mean

$$\tilde{E}(X_{gen}, \mathcal{I}') := |X_{gen}|^{-1} \sum_{x\in X_{gen}} E(x, \mathcal{I}'). \tag{5.25}$$

Adaptation of Diffusivity The diffusivity adaptation step takes as input a quality rating $\tilde{E}(X_{gen}, \mathcal{I}') \in \mathbb{N}$, two parameters $b_1, b_2 \in \mathbb{R}^+$ (controlling the switching behavior from sub-diffusion to super-diffusion) and outputs two adapted parameters $0 < \alpha_1, \alpha_2 < 2$, which according to the power law (5.17) regulate the diffusivity of the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$.

Our aim (as motivated at the beginning of this section) is to adapt the diffusion parameters in such a way that the algorithm automatically focuses its search (by decreasing diffusivity of the generating Lévy flights) when generating high quality (i.e. high coverage) test cases and in turn automatically widens its search (by increasing diffusivity) in

the case of low quality (i.e. low coverage) test cases. As discussed in Section 5.3 we can control diffusivity by setting suitable values of α_1 and α_2. Smaller diffusivity parameters result in frequent long flights and super-diffusion whereas higher parameters reveal frequent small steps and sub-diffusion. To achieve this we select a monotonically increasing function $f : \mathbb{R} \to (0, 2)$ with $f(0) \leq \epsilon$ (for $\epsilon > 0$ sufficiently small) and $\lim_{t\to\infty} f(t) = 2$. Any such function will provide self adaptation of diffusivity of the Lévy flights and we simply choose two functions

$$f_i(t) := \frac{2}{1 + e^{b_i - t}},\ i = 1, 2 \tag{5.26}$$

where $b_i \in \mathbb{R}^+$ are fixed parameters that determine at which point within the quality rating spectrum (i.e. at which mean number of newly discovered basic blocks) the search behavior of $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ switches from sub-diffusion to super-diffusion. With this function we adapt diffusivity to

$$\alpha_i = f\big(\tilde{E}(X_{gen}, \mathcal{I}')\big),\ i = 1, 2. \tag{5.27}$$

The next iteration of test case generation is then executed with adapted Lévy flights.

Test Case Update This step takes as input two sets of test cases $X_{old}, X_{gen} \subset \mathcal{I}$ and outputs an updated set of test cases X_{new}. During the fuzzing process we generate a steady stream of new test cases which we directly evaluate with respect to the set of previously generated inputs (as discussed in the quality evaluation step). However, if we archive every single test case and for each generation step evaluate the k_{gen} currently generated new test cases against the whole history of previously generated test cases, fuzzing speed decays constantly with increasing duration of the fuzzing campaign. Therefore we define an upper bound $k_{max} \in \mathbb{N}$ of total test cases that we keep for quality evaluation of new test cases. Small values of k_{max} may cause the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ to revisit already explored input regions without being adapted (by decreasing the parameters α_i) to perform super-diffusion and widen their search behavior. However, this causes no problem due to the Lévy flight hypothesis (discussed in Section 5.1).

The update of X_{old} with X_{gen} simply follows a *first in first out* strategy. Initially if $|X_{old}| + |X_{new}| < k_{max}$ we append all newly generated test cases so that $X_{new} = X_{old} \cup X_{gen}$. Otherwise we first delete the oldest k_{old} entries in X_{old}, where

$$k_{old} = |X_{old}| + |X_{new}| - k_{max}, \tag{5.28}$$

and then take the union.

Joining the Pieces Now that we have presented all individual parts we can combine them. The overall fuzzing algorithm is depicted in Figure 5.2.

The initial seed generation step outputs a non-empty set of test cases $X_0 \subset \mathcal{I}$, two diffusivity parameters α_1 and α_2, and an initial offset q_0. The inputs X_0 are added to the list of test cases X_{all}. Then the fuzzer enters the loop of test case generation, quality

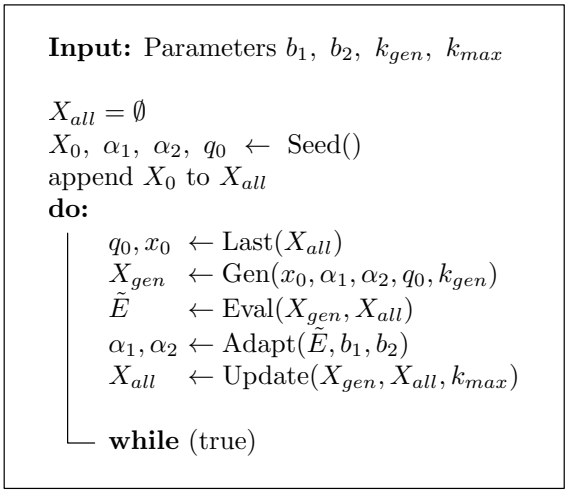


Figure 5.2.: Individual fuzzing algorithm. After initial seed generation the fuzzer enters the loop of test case generation, quality evaluation, adaptation of diffusivity, and test case update.

evaluation, adaptation of diffusivity, and test case update. The first step within the loop (referred to as $Last(X_{all})$) sets q_0 to the last reached offset position of $(L_t^1)_{t\in\mathbb{N}}$. In the first invocation of $Last(X_{all})$ this is simply the already given seed offset, in all subsequent invocations q_0 is updated to the last state of $(L_t^1)_{t\in\mathbb{N}}$. The $Last()$ function also selects the most recently added test case x_0 in X_{all}, which gives the initial condition for $(L_t^2)_{t\in\mathbb{N}}$ in the generation step. In our implementation we realize the $Last()$ function by retaining the reached states of both processes $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ between simulations.

Starting at $L_0^1 = q_0$ and $L_0^2 = x_0(q_0)$ the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ generate the set of new inputs X_{gen} by diffusing through input space with diffusivity α_1 and α_2, respectively. The quality of X_{gen} is then evaluated against the previous test cases in X_{all}. Depending on the quality rating outcome, the diffusivity of $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ is then adapted correspondingly by updating α_1 and α_2 according to the sigmoid functions f_i in Equations (5.26). Then the current list of test cases X_{all} is updated with the just generated set X_{gen} and the fuzzer continues to loop.

Regarding complexity of the fuzzing algorithm we note that all of the individual parts are processed efficiently in the sense that their time complexity is bound by a constant. Especially the evaluation step *Eval()* is designed to scale: In the first iterations of the loop the cost of evaluating X_{gen} against X_{all} is bound by $\mathcal{O}(|X_{all}|^2)$. To counter this growth we defined an upper bound $k_{max} \in \mathbb{N}$ for $|X_{all}|$ in the *test case update* step above.

5.6. Lévy Flight Swarms

Now that we have constructed an individual fuzzing process, we can aggregate multiple instances of such processes to *fuzzing swarms*. Each individual basically performs the

search algorithm described in Section 5.5, but receives additional information from its neighbors and adapts accordingly. Adaptation rules are inspired by social insect colonies [77] and provide a flexible, robust, decentralized, and self-organized swarm behavior as described in Section 5.1.

With the probability spaces $(\Omega_i, \mathcal{F}_i, P_i)$ $(i = 1, 2)$ as defined in Section 5.3, let

$$\mathcal{F}_1 \otimes \mathcal{F}_2 = \sigma(\mathcal{F}_1 \times \mathcal{F}_2) \tag{5.29}$$

denote the σ-algebra generated by the cartesian product $\mathcal{F}_1 \times \mathcal{F}_2$, i.e. the smallest σ-algebra which contains the sets in $\mathcal{F}_1 \times \mathcal{F}_2$. The flight of an individual fuzzer

$$(F_t)_{t\in\mathbb{N}} := (L_t^1, L_t^2)_{t\in\mathbb{N}} \tag{5.30}$$

is formally defined on the product space

$$(\Omega_1 \times \Omega_2, \mathcal{F}_1 \otimes \mathcal{F}_2, P_1 \times P_2), \tag{5.31}$$

where $P_1 \times P_2$ denotes the corresponding product measure. We can then define a swarm S of d individual flights

$$S := \{F^i, \mid i = 1, ..., d\} \tag{5.32}$$

each of which performs the loop of test case generation, quality evaluation, adaptation of diffusivity, and test case update as described in Section 5.5. For each loop iteration, the individuals F^i of the swarm S generate test cases

$$X_{gen}^i := \{x_1^i, ..., x_{k_{gen}}^i\} \tag{5.33}$$

and each individual maintains its own version of aggregated test cases X_{all}^i.

To perform collective fuzzing there are several possibilities for the individuals F^i of the swarm S to exchange information. One strategy would be to define a shared set of already generated test cases $\bigcup_i X_{all}^i$ which could be seen as a global shared memory of already generated test cases. To keep the cost of each evaluation step *Eval()* low, we defined an upper bound $k_{max} \in \mathbb{N}$ for $|X_{all}^i|$ in Section 5.5. If all swarm individuals add their generated test cases to the global shared memory, this would result in a high cost for each individual to evaluate their newly generated test cases X_{gen}^i against $\bigcup_i X_{all}^i$, since the complexity of *Eval()* is bound by $\mathcal{O}(|\bigcup_i X_{all}^i|^2)$.

Therefore, we explore another strategy to share information between swarm individuals. Intuitively, after a fixed amount of search time each F^i of the swarm S receives the actual quality evaluation $\tilde{E}$ of its neighbors and jumps to the one neighbor which is currently searching the most promising input area. If an individual $F_\lambda \in S$ is searching an input area of highest quality $\tilde{E}_\lambda$ test cases among its nearby swarm individuals, all neighbors with lower values of $\tilde{E}_\lambda$ jump to the current position of F_λ in input space. We will formalize this idea in the following, where the index λ refers to local maxima of test case quality $\tilde{E}$.

We first need a metric in input space in order to consider neighborhoods of swarm individuals. As a natural metric in the space of all possible inputs $\mathcal{I} = \{0, ..., 2^N\}$ we

choose the Hamming distance δ: two bit strings $x = (x_1, ..., x_N)$ and $x' = (x'_1, ..., x'_N)$ of size N then have distance

$$\delta(x, x') := \big|\{j \in 1, ..., N \mid x_j \neq x'_j\}\big|. \tag{5.34}$$

We can then simply measure the distance $\delta_S(F_t^i, F_t^j)$ of two individuals

$$F^i = (L^{1,i}, L^{2,i}) \in S \tag{5.35}$$

$$F^j = (L^{1,j}, L^{2,j}) \in S \tag{5.36}$$

at time $t \in \mathbb{N}$ with

$$\delta_S(F_t^i, F_t^j) := \delta(x_t^i, x_t^j) \tag{5.37}$$

where

$$x_t^i = R\big(x_{t-1}^i, L_{t-1}^{1,i}, L_t^{2,i}\big) \tag{5.38}$$

$$x_t^j = R\big(x_{t-1}^j, L_{t-1}^{1,i}, L_t^{2,i}\big) \tag{5.39}$$

are defined as in Equation (5.22). In words, the distance $\delta_S(F_t^i, F_t^j)$ of two swarm individuals $F^i, F^j \in S$ at a certain time $t \in \mathbb{N}$ is the Hamming distance of the respectively two test cases generated at time t.

With this metric we could proceed with considering the R-neighborhood

$$U_R(F_0) := \{F \in S \mid \delta_S(F_0, F) < R\} \tag{5.40}$$

of a swarm individual $F_0 \in S$ for an arbitrary $R \in \mathbb{N}$. However, this definition of neighborhood would result in high processing costs for large swarms: each individual must calculate the distances to all other individuals of the swarm before jumping to the position of the neighbor individual which generated test cases of highest quality $\tilde{E}_\lambda$. Therefore, we introduce a more lightweight method of calculating neighborhoods that scales to large swarms. We periodically divide the whole swarm S into k clusters using a k-means clustering algorithm to yield the disjoint partition $S = \dot{\bigcup}_k C_k$. Each individual $F^i \in C_j$ then only takes into account the test case quality $\tilde{E}$ of individuals within the same cluster C_j before relocation.

The overall swarm fuzzing algorithm is depicted in Figure 5.3. The first part initializes the d swarm individuals F^i $(i = 1, ..., d)$. Each of the d initializations in the first *for* loop basically corresponds to the single fuzzer setup described in Section 5.5, with the minor formal difference that the *Init*() function randomly selects d inputs $x_0^i \in X_0^i \subset I$ $(i = 1, ..., d)$ among the seed input sets to fix the starting points of the F^i.

The algorithm then enters the main *do-while* loop, which consists of three parts: fuzzing, clustering, and relocation. First, all F^i $(i = 1, ..., d)$ start fuzzing the binary performing test case generation, quality evaluation, adaptation of diffusivity, and test case update as described in Section 5.5.

Second, the *Cluster*() function divides the swarm S into k clusters C_j $(j = 1, ..., k)$ as described above. We refer to a single cluster as the *neighborhood* of the swarm individuals

Input: Parameters $d,\ k,\ b_1,\ b_2,\ k_{gen},\ k_{max}$

for $i = 1, ..., d$ **:**
- $X^i_{all} = \emptyset$
- $X^i_0,\ \alpha^i_1,\ \alpha^i_2,\ q^i_0 \leftarrow$ Seed()
- append X^i_0 to X^i_{all}
- $x^i_0 \leftarrow \text{Init}(X^i_0)$

do:
- **for** $i = 1, ..., d$ **:**
  - $X^i_{gen} \leftarrow \text{Gen}(x^i_0, \alpha^i_1, \alpha^i_2, q^i_0, k_{gen})$
  - $\tilde{E}^i \leftarrow \text{Eval}(X^i_{gen}, X^i_{all})$
  - $\alpha^i_1, \alpha^i_2 \leftarrow \text{Adapt}(\tilde{E}^i, b_1, b_2)$
  - $X^i_{all} \leftarrow \text{Update}(X^i_{gen}, X^i_{all}, k_{max})$
- $C_1, ..., C_k \leftarrow \text{Cluster}(x^1_{k_{gen}} \in X^1_{gen}, ..., x^d_{k_{gen}} \in X^d_{gen})$
- $x^1_0, q^1_0, ..., x^d_0, q^d_0 \leftarrow \text{Relocate}(C_1, ..., C_k)$
- **while** (true)

Figure 5.3.: Swarm fuzzing algorithm. The swarm of fuzzers enters the loop of individual fuzzing, clustering with k-means, and relocation of individuals to positions of highest test case quality within respective clusters.

belonging to this cluster. Swarm individuals F^i mutating on nearby inputs (measured with the Hamming metric) are assigned to the same cluster, whereas distant populations share different neighborhoods.

Third, all swarm individuals F^i within the same neighborhood C_j are relocated to the most promising nearby search position. For each cluster C_j $(j = 1, ..., k)$ the *Relocate*() function compares the current test case quality $\tilde{E}^i$ of all F^i within the same neighborhood. Without loss of generality there is one swarm individual $F^j_\lambda \in C_j$ in each neighborhood C_j with maximal quality evaluation $\tilde{E}^j_\lambda$ (in the case of multiple neighbors having the same $\tilde{E}$ we simply could choose one of them randomly). Then the *Relocate*() function resets the initial positions of all $F^i \in C_j$ to

$$L^{1,i}_0 \leftarrow q^\lambda \text{ and} \tag{5.41}$$

$$L^{2,i}_0 \leftarrow x^\lambda(q^\lambda), \tag{5.42}$$

where

$$L^{1,\lambda}_0 = q^\lambda \text{ and} \tag{5.43}$$

$$L^{2,\lambda}_0 = x^\lambda(q^\lambda), \tag{5.44}$$

are the Lévy flight positions of the neighbor individual

$$F^j_\lambda = (L^{1,\lambda}, L^{2,\lambda}) \in C_j \tag{5.45}$$

with currently best test case quality evaluation $\tilde{E}^j_\lambda$ among neighbors in C_j, $(j = 1, ..., k)$.

5.7. Implementation

To show the feasibility of our approach we implemented a prototype for the proposed self-adaptive fuzzing algorithm (as depicted in Figure 5.2). Our implementation is based on Intel's dynamic instrumentation tool Pin [52] to trace the reached basic blocks of a generated test case. In order to calculate the number $E(x_0, \mathcal{I}')$ of newly discovered basic blocks executed by a test case x_0 as defined in Equation (5.14) we switch off *Address Space Layout Randomization* (ASLR) during testing. For developing exploits based on a malicious input x_0 ASLR should naturally be enabled again.

Initially, we simulated the Lévy flights in the statistical computing language R [81] but then changed to a custom sampling method purely written in Python. We construct Lévy flights by summing up independent and identically distributed random variables as indicated in Equation (5.5). Each addend is distributed according to a power law as defined in Equation (5.12). We realize this by applying the *inverse transform sampling* method, also referred to as *Smirnov transform*. The Python script further performs evaluation of the current path exploration performance by direct comparison of executed basic block addresses received from dynamic instrumentation.

We implemented fuzzing swarms by parallel execution of multiple individual fuzzers which are clustered and relocated according to the algorithm described in Section 5.6. For clustering, we apply the Lloyd k-means algorithm.

In our implementation we omit the first step $Last(X_{all})$ within the loop and instead always keep the last reached positions of the processes $(L_t^i)_{t\in\mathbb{N}}$ $(i = 1, 2)$ between simulations. This is due to the construction of new test cases in Equations (5.20)-(5.23) so that the last test case within X_{all} is simply the most recently generated $x_{k_{gen}}$ which will be used as starting position within the subsequent loop iteration. Therefore it suffices to stop the Lévy flights after k_{gen} steps, save their current position, and proceed with adapted diffusivity parameters in the subsequent invocation of the *Gen()* function.

5.8. Discussion

In this section we discuss properties, possible modifications, and expansions of our proposed fuzzing algorithm.

As demonstrated in Section 5.5 our algorithm is self-adaptive in the sense that it automatically focuses its search when reaching high quality regions in input space and widens exploration in case of low quality input regions. One possible pitfall of such a self-adaptive property is the occurence of attracting regions: If the Lévy flights $(L_t^i)_{t\in\mathbb{N}}$ $(i = 1, 2)$ enter regions of high quality and get the response from the quality evaluation step to focus their search (by decreasing their diffusivity), an improper quality rating mechanism might cause the Lévy flights to stay there forever. However, our evaluation method (as defined in Section 5.4) avoids this by favoring test cases that lead the target binary to execute undiscovered basic blocks and in turn devaluates inputs that lead to already known execution paths. Therefore, if the test case generation module gets feedback that it is currently exploring a region of high quality it focuses its search as

long as new execution paths are detected. As soon as exploration of new execution paths stagnates, the feedback from the evaluation module switches to a low rating. Such a negative feedback again increases diffusivity according to Equations (5.26) and (5.27), which again causes the processes $(L_t^i)_{t\in\mathbb{N}}$ $(i = 1, 2)$ to diffuse into other regions of the input space.

Our swarm algorithm for multiple individual fuzzers in Section 5.6 is designed to be flexible, robust, decentralized, and self-organized. The fuzzing swarm is *flexible* in the sense that it adapts to perturbations caused by the nature of Lévy flights and the targeted binary: if an individual fuzzer enters super-diffusion and performs frequent large steps, it simply gets assigned to a new neighborhood in the next clustering step. The swarm is *robust* in the sense that it can deal with loss easily: if an individual fuzzer gets stuck because the target crashed, the swarm algorithm simply omits this individual in the next clustering step. While clustering and relocation is realized by a central component, all individual fuzzers are independent stochastic processes F^i $(i = 1, ..., d)$ which evolve *decentralized*. Finally, paths to bugs in the target emerge *self-organized* during the fuzzing process and are not predefined in any way. While all fuzzers in this chapter are of the same type, we introduce an approach for heterogenous colonies of fuzzers in Section 7.

One main modification of our algorithm (for individual fuzzers) would be interchanging the aim of maximizing code coverage with an adequate objective. In Section 5.4 we defined a quality measure for generated test cases based on the number of new basic blocks we reach with those inputs. Although this is the most common strategy when searching for bugs in a target program of unknown structure, we could apply other objectives. For example, we could aim for triggering certain data flow relationships, executing preferred regions of code, or reach a predefined class of statements within the code. Our fuzzing algorithm is modular and flexible in that it allows to interchange the quality measure according to different testing objectives. More examples of such testing objectives are discussed in the field of test case prioritization (e.g. in [64] and [65]).

5.9. Conclusion

Inspired by moving patterns of foraging animals we introduce the first self-adaptive fuzzer based on Lévy flights. Just like search patterns in biology have evolved to optimal foraging strategies due to natural selection, so have evolved mathematical models to describe those patterns. Lévy flights are emerging as successful models for describing optimal search behavior, which leads us to their application of hunting bugs in binary executables. By defining corresponding stochastic processes within the input space of the program under test we achieve an effective new method for test case generation. Further, we define an algorithm that dynamically controls diffusivity of the defined Lévy flights depending on actual quality of generated test cases. To achieve this we construct a measure of quality for new test cases that takes already explored execution paths into account. During fuzzing the quality of actually generated test cases is constantly forwarded to the test case generating Lévy flights. High quality test case generation with respect to path coverage causes the Lévy flight to enter sub-diffusion and focus its search

on nearby inputs, whereas a low quality rating results in super-diffusion and expanding search behavior. This feedback loop yields a fully self-adaptive fuzzer. Inspired by the collective behavior of certain animal colonies we aggregate multiple individual fuzzers to a fuzzing swarm which is guided by simple rules to reveal flexible, robust, decentralized, and self-organized behavior. Our proposed algorithm is modular in the sense that it allows integration of other fuzzing goals beyond code coverage, which is subject to future work.

6. Triggering Vulnerabilities Deeply Hidden in Binaries

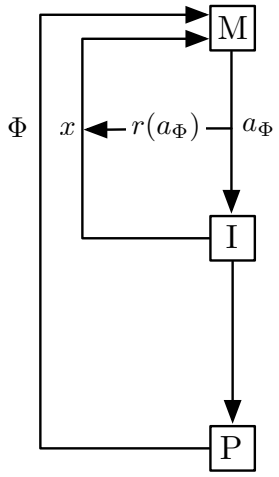


Figure 6.1.: Fuzzing with symbolic reasoning.

In this chapter we upgrade the purely stochastic processes of our model with formal methods in order to get the best characteristics of both worlds: Stochastic search strategies with well known behavior and the precision of symbolic execution that is needed to create partially correct inputs and pass the parsing layers of the target.

We introduce a method for triggering vulnerabilities in deep layers of binary executables and facilitate their exploitation. In our approach we combine dynamic symbolic execution with fuzzing techniques. To maximize both the execution path depth and the degree of freedom in input parameters for exploitation, we define a novel method to assign probabilities to program paths. Based on this probability distribution we apply new path exploration strategies. This facilitates payload generation and therefore vulnerability exploitation. We evaluate our implementation on an OpenSSL X.509 certificate parser and show the practical efficiency of our approach.

The overall approach of this chapter is depicted in Figure 6.1. The mutator engine M, input I, and target program P are connected via rewards, states, and actions as introduced in Chapter 4 of Part I of this work. The mutator engine M performs actions $a_\Phi \in \mathcal{A}$ that combine stochastic fuzzing with computation of certain constraints Φ. The reward $R(x, a)$ as introduced in Equation 4.5 on page 32 is directly inferred from these

actions according to

$$R(x, a) = E(x) + G(a) = E(x) = r(a_\Phi). \tag{6.1}$$

The constraints Φ are derived from symbolic execution of the target, as explained in detail in the following.

6.1. Motivation

Based on our mathematical model developed in Part I of this work, in Chapter 5 we were able to translate important search characteristics of specific stochastic processes into efficient algorithms. However, pure stochastic fuzzing has limitations in many situations common in input parsing. To illustrate such a case, consider the following code snippet:

```
#include <stdint.h>
...
int check( uint64_t num ){
        if( num == UINT64_C(0) )
                assert( false );
}
```

If we want to reach the assertion in the `check` function with a random choice of the integer `num`, we have a probability of 2^{-64} for each try to pass the `if` statement. The situation gets even worse if there are multiple such checks, e.g. in the calculation of a checksum or character match during input parsing. Such code areas are very hard to be passed by pure random input generation and code regions beyond such examples are most likely not covered by fuzzing. In the following we will refer to such cases as *fuzzing walls*. However, the `false` assertion in the above code listing can easily be reached with concolic execution, as the comparison to zero directly translates to a simple expression for the SMT solver. We target realistic examples of fuzzing walls in the context of an OpenSSL X.509 certificate parser with our implementation in Section 6.4.

In this chapter we introduce a new method combining symbolic execution and random testing. Our goals are (1) code coverage in deep layers of targeted binaries which are unreachable by current technologies and (2) maximal degree of freedom in the input variables when discovering a program error.

Before we present the main idea of our approach and the summary of our contributions, we give some background on concolic execution. We especially highlight limitations of concolic execution when applied isolated and motivate a combination of this method with fuzzing as a promising new strategy.

Concolic Execution The main idea of symbolic execution (introduced in [82] and [83]) is to assign symbolic representations to input variables of a program and generate formulas over the symbols according to the transformations in the program execution. Reasoning about a program on the bases of such symbolic representations of execution paths can provide new insight into the behavior of the program. Besides program verification,

symbolic execution nowadays has its biggest impact in program testing. The original idea was extended over the years and developed into concrete symbolic (concolic) execution (see [84] for a detailed introduction). In concolic execution all program variables in scope are represented symbolically. The program is initially executed with arbitrary concrete input values and symbolic constraints over the symbols are generated along the program execution path. Next, one of the collected branch conditions is negated and together with the remaining constraints given to an SMT solver. In [84] the last branch is negated, resulting in a (bounded) depth-first exploration of the execution graph. The authors of [84] also mention that alternative strategies for exploring new paths and thereby covering the execution graph could be applied, e.g. randomly choosing among the collected branch conditions to be negated next or a width-first strategy. Either way the choice of exploration strategy is made, the solution (also called *model*) generated by the SMT solver is injected as new input into the program, which now takes the branch alternative when executed. This is because the SMT solver just calculated the solution of the negation of the former branch constraint so that the newly generated input follows the alternative path. This procedure is iteratively repeated until a halt condition is reached. In the best case the reached halt condition resembles full path coverage of all alternative paths of the program, in the worst case the halt condition is caused by an overloaded SMT solver. The latter is a natural consequence of the exponential growth of the number of paths we have to deal with, which we refer to as the *path explosion* problem.

Concolic execution is more powerful than traditional symbolic execution especially in code regions where pure symbolic reasoning is ineffective or even infeasible. This is often the case for complex arithmetic operations, pointer manipulations, calls to external library functions, or system calls. Consider for example a statement involving a hash function h, e.g. $y = h(x)$. It is impossible to symbolically reason about the involved variables x and y. A SMT solver will not be able to find a satisfying solution for this constraint (see [85] for more details regarding this example). In such cases concolic execution guarantees that execution paths are taken by concrete values and symbolic constraint generation is continued subsequently.

Pure concolic execution, however, has strong limitations. Current SMT solvers are very limited in the number of variables and constraints they can handle efficiently so that concolic execution gets stuck in very early stages of the program. Despite huge advances in the field of SMT solvers (see [86] for a comprehensive overview), concolic execution of large programs is infeasible and in practice will only cover limited parts of the execution graph, e.g. input parsing. The major part of graph coverage must therefore be done with fuzzing.

The Hybrid Approach As we just showed, critical limitations of fuzzing can be overcome with concolic execution, and in turn fuzzing scales much better to path explosion than SMT solvers do. The natural next step is to combine both methods. The idea is to apply concolic execution whenever fuzzing saturates (i.e. stops exploration at a fuzzing wall), and in turn switch back to fuzzing whenever the fuzzing walls are passed by concolic

execution.

However, we still have to deal with the problem of path explosion and therefore still may end up covering only the first execution layers of a program. In the following, we refer to *path depth* as the number of branches along that path, which directly corresponds to the number of basic blocks. Even in the combined approach we are confronted with two challenges. First, if we want to fuzz deep areas of a program, we have to find a way to construct execution paths into such areas and somehow delay path explosion until we have found such a tunnel. Second, to generate a payload and exploit a detected vulnerability in the program under test, we not only have to reach the bug with with a single suitable input, but we have to reach it with maximal degree of freedom in the input values. To be more precise, if we reach a vulnerability with exactly one constellation of the input variables, we most probably would not be able to exploit it in a meaningful way because any attempt to generate a payload (and thereby change the input variables) would lead the input to take a different path in the execution graph. Therefore, we propose a way to maximize the degree of freedom regarding input variables. This yields both, alleviation of vulnerability exploitation and execution paths that reach into deep layers of the program.

Our Contributions In summary, we make the following contributions:

- We propose a new search heuristic that delays path explosion effectively into deeper layers of the tested binary.
- We define a novel technique to assign probabilities to execution paths.
- We introduce DeepFuzz, an algorithm combining initial seed generation, concolic execution, distribution of path probabilities, path selection, and constrained fuzzing.
- We evaluate an implementation of DeepFuzz on an OpenSSL X.509 certificate parser.

The remainder of this chapter is organized as follows. In Section 6.2 we discuss related work. We introduce the DeepFuzz algorithm in Section 6.3. To demonstrate the feasibility of our approach, we implement and evaluate our prototype in Section 6.4, where we apply DeepFuzz to a service that initially parses OpenSSL X.509 certificates. We discuss possible expansions and limitations in Section 6.5. The chapter concludes with a short outlook on further applications in the field of automated exploit generation in Section 6.6.

6.2. Related Work

Symbolic execution has experienced significant development since its beginnings [82, 83] in the seventies to the advanced modern variants invented for program testing in recent years. Especially the last decade has seen a renewed research interest due to powerful Satisfiability Modulo Theory (SMT) solvers [86] and computation capabilities that have

led to advanced tools for dynamic software testing [50, 87, 84, 88, 89]. Cadar et al. [90] and Păsăreanu [91] give an overview of the current status of dynamic symbolic execution. One of the most important variants of dynamic symbolic execution was introduced in [88] and [84], where symbolic constraints are generated along program execution paths of concrete input values. We apply this so-called concolic execution method in parts of our proposed approach.

As discussed at the beginning of this chapter, both concolic execution and fuzzing have severe limitations when aiming for code coverage. Since those limitations are partly complementary to each other, a fusion of concolic execution and fuzzing emerges as natural approach. Majumdar et al. [92] made a first step into this direction by proposing hybrid concolic testing: by interleaving random testing with concolic execution the authors of [92] increase code coverage significantly. However, major questions in this hybrid approach are left open. First, the methods in [92] are based on the CUTE [88] tool, which requires the source code of tested programs and is restricted to programs written in C and the sequential subset of Java. In contrast, we need nothing else than the binary executable under test (compiled from sources of arbitrary high level languages). Second, it is still an open question how to efficiently generate restricted inputs for random testing. We propose a powerful solution for high frequency test case generation that scales to large sets of constraints. Third and most important, the authors of [92] formulate test goals on a rather general level (e.g. maximal branch coverage being one goal). In contrast to this we focus on exploitable vulnerabilities and introduce algorithms that maximize the degree of freedom regarding input variables to achieve both, alleviation of vulnerability exploitation and execution paths that reach into deep layers of the program. In our input maximization algorithm we assign probabilities to program paths in a novel way, which has no counterpart in related work. Although the authors of [93, 94] and [95] also propose assertion of probability weights to paths in the execution graph, they differ significantly in their proposed methods which are based on path condition slicing and computing volumes of convex polytopes.

Closely related to our approach is Driller by Stephens et al. [96] who also combine fuzzing with selective concolic execution in order to reach deep execution paths. Driller switches from pure fuzzing to concolic execution whenever random testing saturates, i.e. gets stuck at a fuzzing wall. To keep the load for symbolic execution low while simultaneously maximizing the chance to pass fuzzing walls with concolic execution, Driller also selects inputs. This selection privileges paths that first trigger state transitions or first reach loops which are similarly iterated by other paths. In contrast, we systematically assign probabilities to paths based on SMT solving performance and select paths according to this probability distribution.

We demonstrate the feasibility of DeepFuzz with an evaluation on an OpenSSL X.509 certificate parser. The authors of [97] and [98] pursue a similar goal by mutating large sets of certificates to test certificate parsing, however their methods are more related to pure fuzzing.

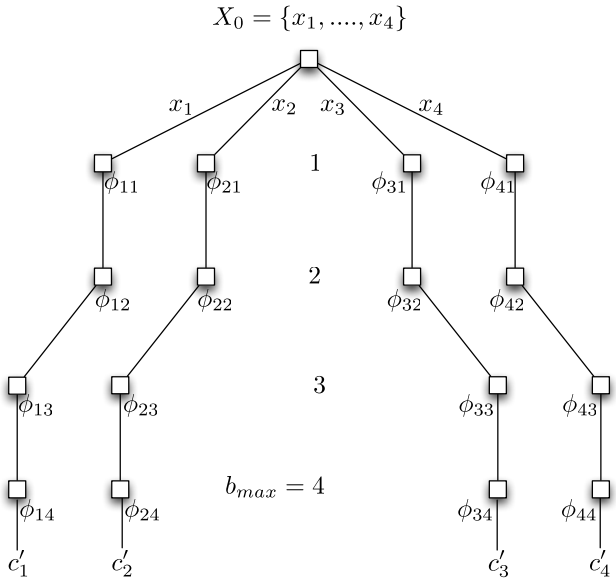


Figure 6.2.: Execution paths c'_i $(i = 1, ..., 4)$ for the initial seed X_0.

6.3. The DeepFuzz Algorithm

In this section we present the DeepFuzz algorithm in detail. The main idea is interleaving concolic execution with constrained fuzzing in a way that allows us to explore paths providing maximal input generation frequency. We achieve this by assigning weights (corresponding to fuzzing performance) to the explored paths after each concolic execution step in order to select the ones with highest probability. In the following, we first describe the individual building blocks, namely initial seed generation, concolic execution, distribution of path probabilities, path selection, and constrained fuzzing. Next, we combine these parts in the overall DeepFuzz algorithm.

6.3.1. Initial Seed Generation

Initially we start with a short period of concrete input generation for the subsequent concolic execution. If the inputs belong to a predefined data format, we generate inputs according to the format definition (as in generational fuzzing). If there is no format specified or available we just generate random input seeds. We denote the set of all possible concrete input values as X and the initial seeds generated in this initial step as $X_0 \subset X$.

To illustrate this, Figure 6.2 shows the execution paths c'_i $(i = 1, ..., 4)$ that an exemplary program takes if the respective inputs $X_0 = \{x_1, ..., x_4\}$ are be passed to it. Each square indicates a possible branch along the path.

6.3.2. Concolic Execution

The concolic execution step receives a set of concrete program inputs $X_{seed} \subset X$ and outputs a set of symbolic constraints collected along the paths belonging to these inputs. At the beginning, directly after the initial seed generation step, we set $X_{seed} = X_0$. The symbolic expressions are basically generated as described at the beginnning of this chapter. However, we adapt the path search heuristics to our approach in a similar way as introduced in [87]. We conduct concolic execution of the program with each input $x_i \in X_{seed}$ until one of the following two halt conditions occur: either the program reaches the predefined goal, which in our case is basically an unexpected error condition, or the number of newly discovered branches taken exceeds a fixed maximum $b_{max} \in \mathbb{N}$.

To keep the notation as clear as possible, in the following we assume without loss of generality that the halting conditions are reached after exactly b_{max} branches. Let c'_i denote the execution path belonging to input x_i and $n' = |X_{seed}|$ denote the number of inputs in X_{seed}. For each branch $j \in \{1, ..., b_{max}\}$ there is a sub-path c'_{ij} which equals c'_i until branch number j is reached. Clearly, the c'_{ij} are sub-paths of c'_i. For each $i = 1, ..., n'$ and $j = 1, ..., b_{max}$ we store the logical conjunction of the negated branch condition λ_{ij} (corresponding to branch number j of execution path c'_i) and the path condition ρ_{ij} of the sub-path c'_{ij} leading to this branch, which yields the $n' * b_{max}$ expression sets

$$\phi_{ij} := \neg\lambda_{ij} \wedge \rho_{ij}. \tag{6.2}$$

With this notation, concolic execution of the input set X_{seed} yields the total set of constraints

$$\Phi := \{\phi_{ij} \mid i = 1, ..., n',\ j = 1, ..., b_{max}\}. \tag{6.3}$$

For each element in Φ the SMT solver checks if the the symbolic constraints are satisfiable and in that case computes a new input x_{ij} for each element $\phi_{ij} \in \Phi$. These newly generated inputs x_{ij} drive the program execution along the original paths c'_i until branch number j is reached and then takes the alternative. We denote these new explored paths as c_{ij}. In the next step we assign probabilities to these paths. To maintain a clear notation and avoid too many indices we work with the union set

$$C := \{c_1, ..., c_n\} := \bigcup_{i,j} c'_{ij}. \tag{6.4}$$

This situation is illustrated in Figure 6.3 where we set $b_{max} = 4,\ n' = 4$ and inputs x_{ij} lead to execution of the paths c_{ij} $(i = 1, ..., n',\ j = 1, ..., b_{max})$.

6.3.3. Distribution of Path Probabilities

Next, we describe our approach to assign probabilities to program paths. This step takes as input a set of paths C and outputs a probability distribution on this set.

One possible strategy is to calculate the cardinality $|I_i|$ of the set of solutions (i.e. models) I_i for the path constraint $\phi_i \in \Phi$ corresponding to c_i and then define weights on

the paths according to number of inputs that travel through it. This strategy is chosen and comprehensively described in [94], where the purpose of assigning probabilities to paths is to provide estimates of likelihood of executing portions of a program in the setting of general software evaluation. In contrast to this we are interested in deep fuzzing and therefore must guarantee maximal possible sample generation in a fixed amount of time. To illustrate this more clearly, consider two sets of constraints Φ_A and Φ_B with (non-empty) solution sets A and B. If we are given only the constraints Φ_A and Φ_B and are interested in *some* solutions in A or B, we simply feed an SMT solver with the constraints and receive solutions. However, computing the cardinality $|A|$ and $|B|$ of *all* solutions corresponding to Φ_A and Φ_B (also called the model counting problem) can be significantly more expensive than the decision problem (asking if there is a single solution of the constraints at all). The authors of [94] rely on expensive algorithms for computing volumes of convex polytopes [99, 100] and integrating functions defined upon them [101]. This would yield a theoretical sound distribution of path probabilities, with the disadvantage of extremely low fuzzing performance in our setting. Further, even if cardinality $|A|$ is significantly greater than $|B|$, meaning that Φ_A has much more solutions than Φ_B, computation of B may take much longer than computation of A. In other words

$$\big(|A| > |B|\big) \nRightarrow \big(T(\Phi_A) > T(\Phi_B)\big), \tag{6.5}$$

where $T(\Phi_i)$ is the time it takes an SMT solver to compute *all* solutions corresponding to the constraints Φ_i. To guarantee high frequency of model generation for effective deep fuzzing we have to build our strategy around a time constraint. Therefore, in order to assign probabilities to the paths $c_1, ..., c_n$ we apply another strategy.

For a fixed time interval T_0 let $k_i(\phi_i, T_0)$ denote the number of solutions for constraints ϕ_i that the applied SMT solver finds in the amount of time T_0. Among the paths $c_1, ..., c_n$ we choose the one whose constraints yield - when given to the SMT solver - the maximal number of satisfying solutions in the fixed amount of time T_0. Therefore, we distribute the probabilities $p(c_i)$ belonging to path c_i according to

$$p(c_i) := k_i\left(\phi_i, T_0\right) \left(\sum_{j=1}^{n} k_j(\phi_j, T_0) \right)^{-1} \tag{6.6}$$

for $i = 1, ..., n$. With $\sum_{i=1}^{n} p(c_i) = 1$ this probability distribution is well defined.

6.3.4. Path Selection

Now that we have n explored paths $C = \{c_1, ..., c_n\}$ weighted with probabilities according to Equation (6.6) in the execution graph, our goal in this step is to select the paths that provide us maximal model generation frequency. Such a set of paths will guarantee us efficient fuzzing and maximal degree of freedom for subsequent payload generation in case we detect a vulnerability.

The defined probabilities $p(c_i)$ in Equation (6.6) directly correspond to the performance in computing inputs for subsequent fuzzing. Practical calculation of those probabilities

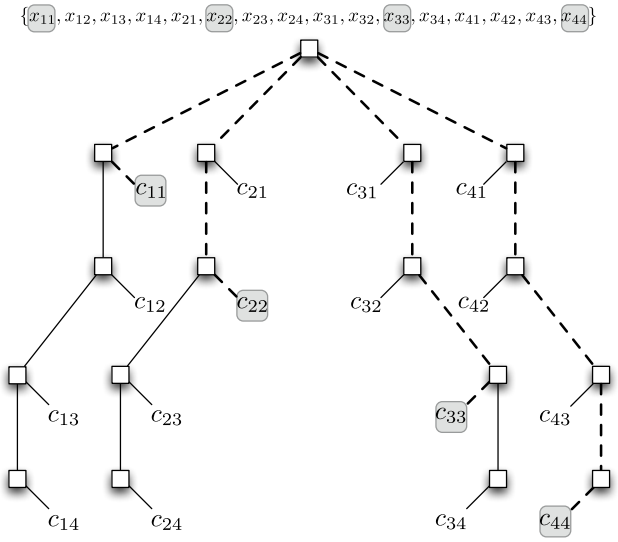


Figure 6.3.: Execution paths c_{ij} for $i = 1, ..., n'$, $j = 1, ..., b_{max}$) and selected set $C_{high} = \{c_{11}, c_{22}, c_{33}, c_{44}\}$.

in an implementation (see Section 5.7) is very efficient: we simply let the SMT solver compute solutions for the path constraints $\Phi_i (i = 1, ..., n)$ in a round-robin schedule and count the number of solutions for each path, which directly yields the probabilities $p(c_i)$. A sufficiently small choice of the computing time T_0 will result in fast path selection. To gain maximal input generation frequency, we could simply choose the single path whose assigned probability is maximal. However, some paths are dead ends and if we would restrict the algorithm to select only a single path for subsequent fuzzing, path exploration might stop too early in some binaries.

Therefore, we select the $m \leq n$ different paths $\tilde{c}_j$ $(j = 1, ..., m)$ with highest probability. In order to make sure that the following path choice is well defined, we prepend a short side note first: It almost never happens in practice that there are two paths assigned with exactly the same probability. If this unlikely situation occurs in practice, we could just randomly choose one among these equiprobable paths and proceed without much changes in the subsequent algorithm. For simplicity of notation we assume without loss of generality that the set $\{p(c_i) \mid i = 1, ..., n\}$ is strictly ordered. We initially choose the path with highest probability

$$\tilde{c}_1 = \underset{c_i \in C}{\arg\max}\ p(c_i) \tag{6.7}$$

and then proceed in the same way

$$\tilde{c}_j = \underset{c_i \in C \setminus \{\tilde{c}_1, ..., \tilde{c}_{j-1}\}}{\arg\max}\ p(c_i) \tag{6.8}$$

until we obtain the path set

$$C_{high} = \{\tilde{c}_j \mid j = 1, ..., m\} \tag{6.9}$$

including the m paths with hightest probability. On the one hand, setting the parameter $m = n$ will result in fast path explosion. On the other hand, setting $m = 1$ might be too restrictive for some binaries. Therefore, we initially set m to a small integer and then run parameter optimization to adapt to the specific binaries in testing experiments.

Path selection is illustrated in Figure 6.3, where we set $m = 4$ and mark an exemplary set of paths with hightest probability

$$C_{high} = \{c_{11}, c_{22}, c_{33}, c_{44}\}, \tag{6.10}$$

indicated with dashed lines. As introduced in Equation (6.4), we refer to these paths as c_1, c_2, c_3, and c_4 for simplicity of notation.

6.3.5. Constrained Fuzzing

Now that we have selected the paths C_{high} with highest probability, we continue with fuzzing deeper layers of the program. Remember we denoted the set of all possible concrete input values as X and the set of inputs belonging to path c_i as $I_i \subset X$ $(i = 1, ..., n)$. To start fuzzing into the program from an endpoint of a selected path $c_i \in C_{high}$, the generated fuzzing inputs have to fulfill the respective path constraints ϕ_i, otherwise they would result in a different execution path. There are basically three possible strategies to generate inputs (i.e. subsets of I_i) that satisfy the respective constraints:

Random generation of inputs with successive constraint filtering This strategy would initially generate a random input set $X_{rand} \subset X$, which would be given to an SMT solver in order to check if a concrete input $x \in X_{rand}$ satisfies the constraint ϕ_i and therefore belongs to I_i. However, filtering the generated inputs in X_{rand} by checking for satisfiability of respective path constraints would most unlikely leave any input over, i.e. $X_{rand} \cap I_i = \emptyset$ with high probability. This is obvious due to the fact that the path constraints in ϕ_i symbolically represent all branch conditions along the path c_i, in particular fuzz-walls (as introduced at the beginning of this chapter). Randomly generating input values that satisfy such a fuzz-wall constraint in ϕ_i is therefore clearly as unlikely as passing such a wall with pure fuzzing.

Pure SMT solver-based input generation With this strategy we would inject all the constraints in ϕ_i into an SMT solver, that in turn computes a set of possible solutions. The problem with this strategy is that an SMT solver is sometimes slow and inefficient in computing solutions and the fuzzing input generation rate would drop significantly. This is due to the fact that an SMT solver cannot effectively handle large amounts of variables constrained in large amounts of equations. For example, consider a situation where the input consists of a large file F and the targeted program only checks a small part F' of it during initial parsing. Using an SMT solver to generate both the constrained part F' *and* the unconstrained part of F would be inefficient. This motivates the third strategy.

Random generation of independent input variables with subsequent constraint solving Here, we randomly generate input values for all variables that are independent (also called *free*) in ϕ_i. An SMT solver subsequently generates a model for the remaining dependent variable constraints.

To illustrate this situation with an example, consider a function which takes $x = (x_1, ..., x_l) \in X$ as input to test a checksum over the first $l-1$ variables in such a way that the check succeeds if $\sum_{i=1}^{l-1} = x_l$, otherwise the program rejects the input. In this case the variables $x_1, ..., x_{l-1}$ are independent in the constraints gathered from the checksum path. Therefore, it is not necessary to feed them into the solver. We can just randomly generate any concrete value for them and let the solver calculate x_l so that the checksum is correct. This guarantees that the SMT solver receives as small as possible number of constraints in order to generate the dependent variables as fast as possible.

In summary, the first strategy is infeasible, whereas strategies two and three are more similar to each other for small input sizes. However, if we deal with larger inputs where only a small minority of input variables are constrained by the current path constraint ϕ_i there is no need to feed a huge amount of path constraints for independent input variables (e.g. a larger parameter file) into an SMT solver. We proceed with the third approach as it guarantees us maximal input generation frequency and scales better to large inputs.

In the following, we refer to the frequency of input generation for path c_i as $f(\phi_i)$. The above reasoning yields

$$f(\phi_i) \geq \frac{k_i(\phi_i, T_0)}{T_0}, \tag{6.11}$$

meaning that the number of models for ϕ_i found by the SMT solver in time T_0 is less or equal than the number of inputs generated with strategy three in time T_0.

6.3.6. Joining the Pieces

Now that we have described all individual parts we can combine them for the the overall DeepFuzz algorithm. After the initial seed generation (SG) is completed we run concolic execution (CE), distribution of path probabilities (DP), path selection (PS), and constrained fuzzing (CF) in a loop until a halt condition is reached. A halt condition is given either if a predefined goal (e.g. a program crash) is reached, of if the constrained fuzzing performance collapses. In the latter case the total number of solutions that the applied SMT solver finds in the fixed amount of time $m * T_0$ drops below a predefined bound k_{min}

$$\sum_{i=1}^{m} k_i(\phi_i, T_0) < k_{min} \tag{6.12}$$

and we leave the loop to procede with solely constrained fuzzing. The overall algorithm is depicted in Figure 6.4. We show the efficiency and feasibility of the DeepFuzz algorithm with our implementation in Section 5.7.

Input: Program P, Parameters $m,\ k_{min},\ T_0,\ b_{max}$

$X_{seed} \leftarrow \text{SG}\ (P)$
do:
  $\Phi = \emptyset$
  $C = \emptyset$
  for each x **in** X_{seed} **do:**
    $c,\ \phi\ \leftarrow \text{CE}\ (x, b_{max})$
    append ϕ to Φ
    append c to C
  $Prob\ \leftarrow\ \text{DP}\ (\Phi, C)$
  $C_{high}\ \leftarrow\ \text{PS}\ (Prob, C)$
  $X_{seed}\ \leftarrow\ \text{CF}\ (C_{high}, \Phi)$
  while $\neg$ condition (11) ; i.e. $\left(\sum_{i=1}^{m} k_i(\phi_i, T_0) \geq k_{min}\right)$
$\text{CF}(C_{high}, \Phi)$

Figure 6.4.: DeepFuzz main algorithm with parameters $m,\ k_{min},\ T_0$, and b_{max}.

In the language of Markov decision processes as introduced in Part I of this work the actions $a_\Phi \in \mathcal{A}$ are given by the constrained fuzzing (CF) function. The reward $R(x, a)$ as introduced in Equation 4.5 on page 32 is given by the path probabilities defined in Equation 6.6.

6.4. Implementation and Evaluation

In the following we present details of our implementation and evaluate DeepFuzz on an OpenSSL X.509 certificate parser to show the feasibility of our approach. We choose the OpenSSL libraries in order to have an evaluation on a real target with a wide range of critical services in the internet today. Certificate parsing based on the widely deployed TLS implementation OpenSSL is a suitable target for evaluation as it provides us a vivid example for the powerful advantages of DeepFuzz. Note that in this section we refer to OpenSSL source code functions in order to illustrate the target behavior, however all results are gathered by applying DeepFuzz directly on the executable parser binary.

Our implementation is based on Intel's dynamic instrumentation tool Pin [52] and the concolic execution framework Triton [102], which itself uses the Z3 SMT solver [103] developed by Microsoft Research. We conduct all tests on an Intel Xeon X5670 (2,93 GHz) with 48 GB RAM.

Our target is a service that parses Base64-encoded X.509 certificates (in `.pem` format) and reacts depending on the parsing results. The program triggers different logging and response routines depending on the certificate's subject and issuer, country code, version, signature algorithm, validity period, and X.509 extensions. Our goal is to build paths with high probability through the certification validation layer of this OpenSSL parser in order to fuzz deeper layers of the unknown program routines.

The parser receives a certificate as input and parses it using common OpenSSL pars-

ing functionality. During the parsing process, routines are executed depending on the certificate's properties. DeepFuzz initially generates a set of seed certificates in the seed generation step and loops through concolic execution, distribution of path probabilities, path selection, and constrained fuzzing until a halt condition is reached and a tunnel into deep layers of the target is constructed. DeepFuzz then switches to purely constrained fuzzing.

6.4.1. Time and Memory Complexity

First, we evaluate the overall performance of our approach in terms of RAM usage M, the time T_{pc} to build the path constraints Φ, and the time T_{solve} to solve these constraints with the SMT solver Z3. Our targeted service initially checks for a match in the certificate's issuer string with several fixed substrings. This subroutine of the certificate parser is a suitable target to measure time and memory complexity. DeepFuzz successively constructs certificates that match these substrings. For each character in the issuer string, DeepFuzz runs the program with a concrete input, builds path constraints that must be guaranteed to fulfill the character match, solves these constraints, injects the newly generated model into a new certificate and reruns the program until it has found the correct substring. Subsequently, DeepFuzz assigns probabilities to the discovered paths, selects C_{high} according to Section 6.3 and then uses the path constraints Φ corresponding to paths in C_{high} for constrained fuzzing into deeper layers of the parser. In the following, we measure path depth in the number of new branches along that path. This measure directly corresponds to the number of newly discovered basic blocks.

Regarding memory complexity Figure 6.5 shows an almost linear dependance of RAM usage on path depth. In our setup this did not cause severe limitations. For large sets of constraints belonging to very deep paths this results in swapping data between RAM and hard disk.

The almost linear dependence of the path constraint generation time T_{pc} adds to the efficiency of DeepFuzz. Path constraints are generated (as described in Section 6.3) only m times every cycle of CE-DP-PS-CF. Therefore, although T_{pc} is much larger than T_{solve}, generating Φ adds only little to the overall time complexity of DeepFuzz (which is true even for larger choices of m). Finally, when the loop CE-DP-PS-CF is left DeepFuzz proceeds with solely constrained fuzzing which is not influenced by T_{pc}.

Finally, Figure 6.5 shows a gradual increase of T_{sove} for very deep paths with around $4{*}10^5$ branches. Growth of model generation time T_{solve} for deep paths directly translates to a decay in fuzzing performance (i.e. input generation frequency). The growth rate of T_{solve} is determined by the program under test and the sort of operations it performs. Depending on the choice of k_{min} this results in switching to pure constrained fuzzing (as described in Section 6.3). The switching condition is determined by equation (6.12) where the choice of parameters T_0 and k_{min} depends on the overall available fuzzing time and computing capability. One significant advantage of our approach is that model generation can be highly parallelized with a computing cluster running multiple distributed instances of Z3. Therefore, constrained fuzzing performance scales to parallelization. Since a moderate increase of T_{solve} can be countered with parallelization, the results suggest

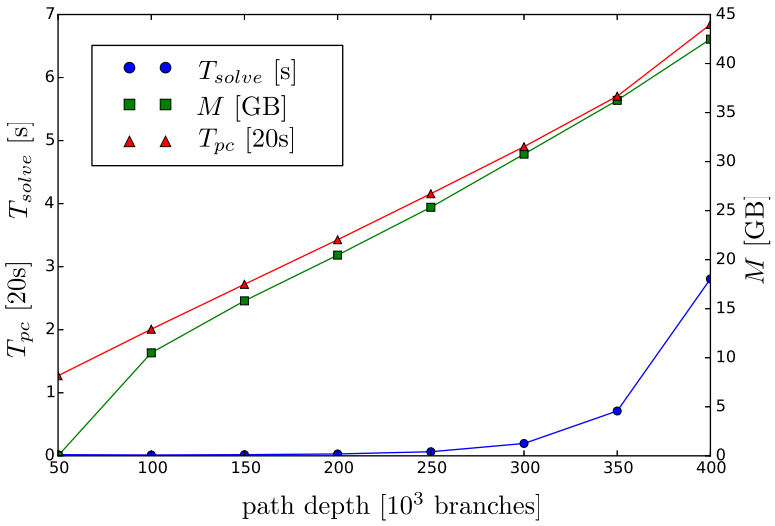


Figure 6.5.: Evaluation of time and memory complexity.

that if we want to apply DeepFuzz on very large targets, we are most likely to switch to a cluster computing setup.

6.5. Discussion

With DeepFuzz we effectively relocate graph coverage to deeper layers of the program. We achieve this by constructing a tunnel into deep layers of the execution graph with subsequent constrained fuzzing through it. Instead of source code instrumentation, we only need compiled binaries for program testing. This is a huge advantage for the same reasons as stated in [104]. First, we are independent on the high level language and build processes. Second, we avoid any problems caused by compiler transformation after the build process, realized for example by obfuscation. Third, DeepFuzz is suited to fuzz closed source targets.

Another important aspect of DeepFuzz is the ability to highly parallelize most parts of the proposed algorithm in Section 6.3. All substeps, i.e. concolic execution, distribution of path probabilities, path selection, and constrained fuzzing can be modularized and distributed for parallel computing with a suitable framework.

One disadvantage of DeepFuzz is that it is not directed towards a tagged point in the execution graph. It builds paths as deep as possible into the program, however with no preferably direction. In order to address this issue we are currently considering how to combine our approach with previous work on driving execution of the input space towards a selected region. Such a directed exploration can be achieved by using fitness functions as introduced in [105, 106], and [107]. For example, we could integrate fitness functions in the path selection step in Section 6.3.

Further, we could improve DeepFuzz regarding fuzzing throughput by taking snapshots

at certain predefined program execution points. For each selected path $c_i \in C_{high}$ we could run the program with corresponding input x_i until the negated branch (as described in the concolic execution step in Section 6.3) is reached. Directly after the alternative is taken we then could generate a program snapshot including all processor registers and the memory state. With $|C_{high}| = m$ we would have to take m snapshots. In the subsequent fuzzing step, we would restore these snapshots and directly start fuzzing at those points. This would allow us to skip the whole program execution before the snapshot, which depending on the current path depth would save a significant amount of time. However, before restoring the snapshot we must conduct taint analysis for all memory areas and registers the input may reach and generate constraints for the tainted areas. The content of each tainted area must then be computed by an SMT solver. It remains an open question if such an approach is an improvement regarding fuzzing throughput or rather a change for the worse.

Finally, our DeepFuzz approach may help to circumvent current bottlenecks related to automatic exploit generation as described by Avgerinos et al. in [108] and [109], where the authors explicitly stress that "programs with deep bugs" are currently not exploitable by their tool.

6.6. Conclusion

We present a powerful approach to trigger vulnerabilities in deep layers of binary executables. DeepFuzz constructs a tunnel into the program by applying concolic execution, distribution of path probabilities, path selection, and constrained fuzzing in a way that allows fuzzing deep and vulnerable areas of the program. This enables us to detect vulnerabilities that are completely out of reach for comparable random testing approaches. We implement and evaluate our proposed algorithm on an OpenSSL X.509 certificate parser. Further, we discuss advantages, current limitations, and possible expansions of DeepFuzz. We assume that DeepFuzz will have impact in the related field of automatic exploit generation, since it solves current bottlenecks in this research area.

7. Guiding a Colony of Fuzzers with Chemotaxis

In this chapter we introduce an elegant way to combine stochastic with formal methods while keeping the overall fuzzing process efficient. As we saw in the last Chapter 6 combining stochastic fuzzing with symbolic combines the best of both worlds, but also comes with a high price: Formal methods are computationally expensive and slow our algorithms down. Larger fuzzing campaigns would therefore greatly benefit, if fast and efficient *worker* fuzzers could run isolated from symbolic execution while being guided by explorers. How can we create an information channel between isolated fuzzing instances in order to allow such strategies? We approach this question via a state synchronization mechanism.

We present a bio-inspired method for large-scale fuzzing to detect vulnerabilities in binary executables. In our approach we deploy small groups of feedback-driven explorers that guide colonies of high throughput fuzzers to promising regions in input space. We achieve this by applying the biological concept of chemotaxis: The explorer fuzzers mark test case regions that drive the target binary to previously undiscovered execution paths with an attractant. This allows us to construct a force of attraction that draws the trailing fuzzers to high-quality test cases. By introducing hierarchies of explorers we construct a colony of fuzzers that is divided into multiple subgroups. Each subgroup is guiding a trailing group and simultaneously drawn itself by the traces of their respective explorers. We implement a prototype and evaluate our presented algorithm to show the feasibility of our approach.

7.1. Motivation

As introduced in Chapter 2 of this work, state-of-the-art fuzzing frameworks all share one overall goal: Generating and pitching suitable program inputs into the target in order to eventually trigger an exploitable bug. For suchlike bug hunting there is a straight forward track: The more input we generate to test a binary target the more code coverage we achieve during program execution and the more likely we will find what we are looking for. This results in parallel large-scale testing by running distributed fuzzer instances on a computer cluster. However, state-of-the-art in distributed large-scale fuzzing basically reduces to pure parallelization. Recent research focuses on advancing single fuzzers and optimal scheduling of fuzzers, test case corpora, and targets during fuzzing campaigns [110]. But how can we optimize the interaction between fuzzers? How can we transform a cluster of isolated fuzzers into a colony that works together and collectively adapts to the binary under test?

Inspired by biology two observations in particular guide our research presented in this section: Colonies with dedicated explorers and the concept of chemotaxis.

Colonies with Explorers Several species such as honeybees, ants, rats, and bats reveal dedicated exploring behavior of colony individuals that primarily function as scouts. Investigation of the environment by just a small fraction of explorers seems to be an efficient way for some colonies to gain information regarding the surrounding territory. In case the explorer found an interesting spot (for example a source of food during foraging) it reports its findings back to the colony. The famous dance of the honeybees [111] is just one example for this behavior. Hence we define dedicated subgroups of explorer fuzzers that guide higher throughput worker fuzzers. In fact, we can divide modern fuzzing frameworks into two categories, namely (1) feedback fuzzers that instrument their targets in order to gain runtime information during program execution and (2) black-box fuzzers that are blind to what happens during execution and only see program crashes in case of a triggered bug. While fuzzers of the first category (including white-box and evolutionary fuzzers) are relatively slow they nowadays achieve similar levels of code coverage compared to traditional fast executing black-box fuzzers. Both categories, the relatively slow feedback driven explorers as well as the fast and efficient black-box worker fuzzers have their right to exist in modern fuzzing campaigns and both provide comparable results. Inspired by colony behavior in biology, is there a way to combine the explorer sight into runtime (gained by dynamic instrumentation) with the speed of black-box worker fuzzers? How can we achieve guidance by the explorers and transfer information to the blind black-box fuzzers? This brings us to the second observation found in biology.

Chemotaxis Regardless if we look at bacteria, mold fungus, termites, ciliates, or algae, all those species have one thing in common: They make use of chemical substances to transmit information between individuals of the colony in order to trigger collective behavior. The movement of organisms responding to chemical stimuli is called *chemotaxis*. Positive chemotaxis causes the individuals to move towards regions of higher concentration of an attractant. Ant colonies [112] coordinating their foraging behavior using attracting trail pheromones impressively illustrate the power of chemotaxis. Can we mimic social behavior of biological colonies using the concept of chemotaxis?

In this chapter, we construct an algorithm for distributed large-scale fuzzing that equips feedback-driven explorer fuzzers with the ability to attract high throughput fuzzers by marking regions in the input space with an attractant. First, we develop the main idea on a single subgroup of explorers guiding a single subgroup of workers: By controlling the attractant concentration among promising test case regions the *seeing* feedback-driven explorers guide the colony of *blind* (but fast) black-box fuzzers in order to maximize code coverage. Second, we generalize this approach to multiple hierarchies of fuzzers: We introduce multiple hierarchies of explorers by further subdividing our scouts according to their overall test case throughput.

In summary, we make the following contributions:

- We introduce a novel method for distributed large-scale fuzzing in computer clusters based on the biological concept of chemotaxis in order to maximize coverage of execution paths in the target under test.

- We construct a mechanism for distributing attractants in input space and define the resulting force field of attraction exerted on high throughput fuzzers.

- We implement and evaluate our presented algorithm to show the feasibility of our approach.

The remainder of this chapter is organized as follows. In Section 7.2 we present our algorithm for guided fuzzing. We implement and evaluate our approach in Section 7.3 and discuss properties, modifications, and expansions of the proposed algorithm in Section 7.4. The chapter concludes with a short outlook in Section 7.5.

7.2. Guided Fuzzing

In this section we present the overall algorithm for collective random testing of binary targets by a colony of fuzzers guided by dedicated explorers.

Our final goal is to optimize massively parallel large-scale fuzzing in computer clusters to find vulnerabilities in a binary target. Let $\dot{F}$ denote the set of feedback-driven fuzzers and F the set of fast non-instrumenting black-box fuzzers, respectively. Inspired by biology we refer to $\dot{F}$ as the explorers and to F as the worker individuals. The explorers receive information from dynamic instrumentation (e.g. regarding code coverage) and therefore see what happens during execution of the target. As motivated in the introduction we present a guidance mechanism that enables the seeing explorers to transfer information to the blind black-box worker fuzzers by mimicking the concept of chemotaxis. We achieve this by constructing explorer traces in the target input space to attract the workers F. In the following we first formalize how to construct such traces and then define the force of attractivity and resulting colony movement analog to chemotaxis.

7.2.1. Attractant Trace Generation

We assume the inputs of the target binary under test to be bit strings of length N and denote the input space as $\mathcal{I} = \{0, ..., 2^N\}$. Each fuzzer provides a corpus $C \subset \mathcal{I}$ of current test cases. During a fuzzing campaign the individual fuzzers constantly update their set of current test cases, which generates a trace in input space for each fuzzer. Inspired by chemotaxis we want the explorers to leave behind an attractant on their way through input space. More formally, assume we have n_E explorers $\dot{F}^i$ each starting with a set of seed inputs $\dot{C}_{t_0}$. After some time t_1 of fuzzing, each $\dot{F}^i$ has updated its initial seed inputs to the current working corpus $\dot{C}^i_{t_1}$. During the fuzzing campaign, the $\dot{F}^i$ generate corpora $\dot{C}^i_{t_1}, \dot{C}^i_{t_2}, ... \subset \mathcal{I}$. To construct a trace of $\dot{F}^i$ in $\mathcal{I}$, we calculate the center of each intermediate corpus of test cases and then mark these centers with an attractant.

Trace Generation We first need to define the center of a corpus $\dot{C} \subset \mathcal{I}$ of test cases. Instead of the arithmetical mean we are interested in the bit string $\hat{c}$ that is most similar to all of the strings in $\dot{C}$. This choice is justified by the following example: Consider a corpus $\dot{C}$ of bit strings each of which respects the input format of a given target. The arithmetical mean of $\dot{C}$ might be a bit string with corrupted file format including wrong headers and metadata. Therefore, we define the center $\hat{c}$ of $\dot{C}$ to be the string of length N that coincides with the majority of inputs in $\dot{C}$ in each bit position. The complexity of this calculation is bound by $\mathcal{O}(n^2)$. Periodically extracting the corpus of current test cases of the explorers $\dot{F}_i$ and calculating their centers $\hat{c}^i$ yields a trace

$$T^i := \left(\hat{c}^i\right)_{\tau \in \mathcal{T}} := \left(\hat{c}^i_{\tau_0}, \hat{c}^i_{\tau_1}, \hat{c}^i_{\tau_2}, ...\right) \tag{7.1}$$

where $\mathcal{T} = \{\tau_0, \tau_1, \tau_2, ...\}$ indicates the extraction times during the fuzzing campaign.

Attractant Spraying Now that we have a trace T^i for each explorer $\dot{F}^i$ we can spray this trace with an attractant in order to draw the black-box worker fuzzers F^i. In this step we augment each center of trace T^i with an attractant concentration that decreases over time. Naturally, the most recently generated corpus of an explorer should have a higher concentration of attractant than a previously generated corpus. This correlates to diffusivity and resulting fall in concentration of real chemical attractants in biological chemotaxis. To realize this we define a monotonically decreasing function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ to yield the sprayed trace

$$\bar{T}^i := \left(\left(\hat{c}^i_{\tau_0}, f(t-\tau_0)\right), \left(\hat{c}^i_{\tau_1}, f(t-\tau_1)\right), ... \right) \tag{7.2}$$

$$= \left(\hat{c}^i_{\tau}, f(t-\tau)\right)_{\tau \in \mathcal{T}} \tag{7.3}$$

for $i = 1, ..., n_E$, where t denotes the current time of the fuzzing campaign. In our implementation (see Section 5.7) we generate the sprayed traces periodically after a fixed amount of time so we can assume without loss of generality the discrete time indexing set $\mathcal{T} = \mathbb{N}$. The choice of spraying function f determines attractant concentration of explorer traces over time. If f decays fast, the explorers will leave only a short attracting trace in time, whereas a slower decay yields longer attracting traces. To avoid persistent attraction of already extensively explored regions we must construct f such that attractant concentration decays to zero after some time, i.e. $\lim_{t\to\infty} f(t) = 0$. Moreover to keep computing complexity in subsequent steps low we define f to map identical to zero after time t_z, i.e.

$$\forall t \geq t_z : f(t) = 0. \tag{7.4}$$

We discuss different choices of f and resulting attracting behavior of the black-box worker fuzzers in Section 5.7.

7.2.2. Positive Chemotaxis

Next, we construct an attraction mechanism for the sprayed traces $\bar{T}^i$ left behind by the explorers $\dot{F}^i$ $(i = 1, ..., n_E)$. The traces $\bar{T}^i$ should attract the black-box worker fuzzers

F^j $(j = 1, ..., n_W)$. Again, we refer to the position of an $\dot{F}^i$ as the center $\hat{c}^i \in I$ of its current corpus of test cases $\dot{C}^i \subset \mathcal{I}$. Mathematical modeling of chemotaxis usually makes use of partial differential equations [113, 114], which describes movement and emerging spatial pattern formation accurately in terms of biology. Since we are more interested in computational efficiency than in biological accuracy, instead of simulating our colony of fuzzers with partial differential equations we define a lightweight attracting function g that acts as a force of gravity on the corpora of black-box fuzzers F^j. While f (as described above) determines the distribution and decay of attractant concentration of traces in input space $\mathcal{I}$, g determines the force of attraction dependent on both the distance and the concentration of the attractant. Therefore, $g : \mathbb{R}^2_{\geq 0} \to [0, 1]$ is a function of two variables. We discuss and evaluate different choices of g and resulting attracting behavior in Section 5.7.

To determine the force of attraction that an explorer trace exerts on a black-box worker fuzzer F^j $(j = 1, ..., n_W)$ we need the attractant concentration of its trace as well as the distance between centers of the trace $\bar{T}^i$ and the corpus C^j of F^j. The individual centers $\hat{c}^i \in \mathcal{I}$ of explorer traces $\bar{T}^i$ have already assigned a concentration as given in Equation (7.2). For the metric we choose Hamming distance δ in $\mathcal{I}$: Two bit strings $x = (x_1, ..., x_N)$ and $x' = (x'_1, ..., x'_N)$ then have distance

$$\delta(x, x') := \left|\{j \in 1, ..., N \mid x_j \neq x'_j\}\right|. \tag{7.5}$$

For a single test sample $x \in \mathcal{I}$ function g then gives the force of attraction $a \in [0, 1]$ that a center $\hat{c}^i$ exerts on x at time t:

$$a = g\left(f(t - \tau_i), \delta(\hat{c}^i, x)\right). \tag{7.6}$$

Now that we have defined the force a of attraction on $x \in \mathcal{I}$, we construct a movement of x analog to chemotaxis. We can move x towards $\hat{c}^i$ in the Hamming distance if we flip bits in x to match the corresponding bits in $\hat{c}^i$. Therefore, let $a \in [0, 1]$ be the fraction of bits in x that we flip to match bit string $\hat{c}^i$, where the bit positions to be flipped are randomly chosen among $1, ..., N$. For example, $a = 1$ causes all bits in the mutated version x' of x to match those in $\hat{c}^i$ resulting in $\delta(\hat{c}^i, x') = 0$. An attracting force of $a = 0$ on the other hand leaves x unchanged.

Finally, an explorer $\dot{F}^i$ draws a black-box worker F^j by letting its trace $\bar{T}^i$ (i.e. all centers $\hat{c}^i$ of its trace with nonzero attractant concentration) simultaneously attract all test cases in the current corpus $C^j \subset \mathcal{I}$ of F^j.

7.2.3. Guided Fuzzing Algorithm

The algorithm for guiding a dedicated colony of black-box fuzzers is depicted in Figure 7.1.

The first two loops initialize the n_E explorers as well as the n_W black-box worker fuzzers. The seed input corpora $\dot{C}^i_{t_0}, C^j_{t_0} \subset \mathcal{I}$ are sets of bit strings of length N. They can be generated randomly or alternatively may originate from a previous fuzzing campaign, but we don't assume any constraints on them (e.g. validity regarding the input format).

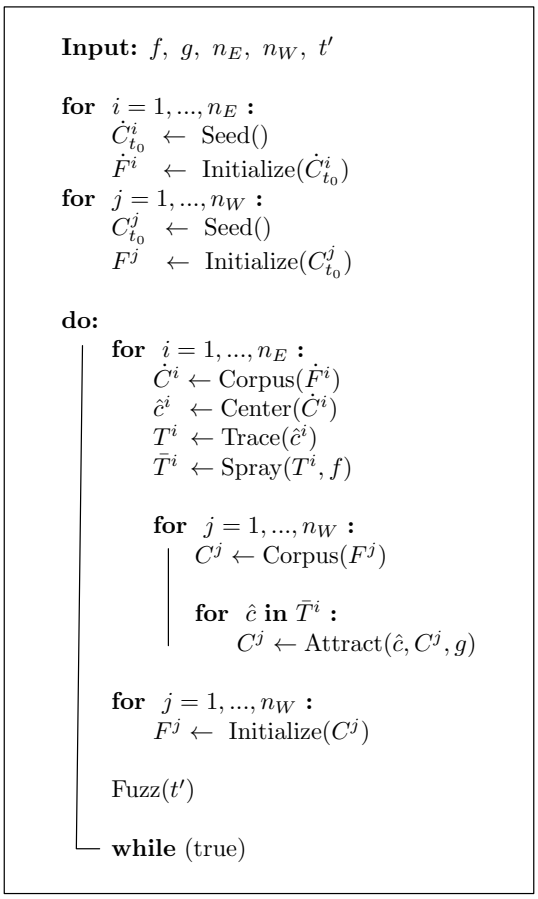


Figure 7.1.: Algorithm for guided fuzzing with input functions f, g and parameters n_E, n_W, and t'.

After initialization phase we enter the process of attractant trace generation, positive chemotaxis, and fuzzing. This main iteration is repeated until a tester stops the fuzzing campaign. The first loop in the main iteration extracts the test case corpora $\dot{C}^i$ of explorers $\dot{F}^i$, calculates their centers $\hat{c}^i$, appends them to the respective traces T^i and sprays the traces with the attractant according to the choice of f. Next, the centers $\hat{c}^i$ of the sprayed traces $\bar{T}^i$ attract all n_W test case corpora C^j of black-box worker fuzzers F^j. This force of attraction results in positive chemotaxis and is regulated by function g as given in Equation (7.6).

Next, we reinitialize the black-box worker fuzzers F^j with the updated respective test case corpora and let the whole colony of fuzzers perform random testing for a fixed amount of time t'.

Regarding computational complexity of the proposed algorithm we constructed each step to be efficient. The cost of center calculation is bound by $\mathcal{O}(n^2)$, which is tractable considering that we only process the working set of current test cases of an explorer. Spraying the traces T^i with an attractant as indicated by Equation (7.2) is a lightweight operation on a two-dimensional array that holds for each calculated center $\hat{c}^i$ the corresponding attractant concentration given by f. Calculation of the force of attraction as defined in Equation (7.6) requires computing the Hamming distance, which requires low overhead. Further, we carefully bound the time for computing the force of attraction that the explorer traces $\bar{T}^i$ $(i = 1, ..., n_E)$ exert on the corpora C^j $(j = 1, ..., n_W)$ by limiting the number of centers with nonzero attractant concentration, as guaranteed by Equation (7.4). Finally, we process these steps that lead to repositioning of the corpora of F^j only once for each time interval t'. During the fuzzing step (denoted by *Fuzz(t')* in Figure 7.1) all fuzzer instances of both the explorers and the black-box workers run unaffected.

7.2.4. Explorer Hierarchies

Next, we generalize the algorithm for guided fuzzing as depicted in Figure 7.1 to multiple hierarchies of explorers by further subdividing our scouts according to their overall test case throughput. This further distinction is motivated by the observation that there is a spectrum between feedback-driven fuzzers and black-box fuzzers. For example, test frameworks enhanced with symbolic execution functionality (such as [17, 50, 87]) are computationally more complex than more efficient evolutionary fuzzers (see [115] for a recent benchmark), but both categories make use of feedback from binary instrumentation. Further, the efficiency of black-box fuzzers depends on the targeted input format: In some cases (of complex input formats) it is useful to deploy a grammar-based fuzzer that generates mostly valid test cases in order to feed them as seed into feedback-driven fuzzers. In order to cover such situations with our approach, we need to define multiple hierarchies of fuzzers. This results in a colony of fuzzers divided into multiple subgroups each guiding a trailing group and simultaneously drawn itself by the traces of their respective explorers.

To achieve this, we divide the whole fuzzing colony into n_K classes $\mathcal{F}^i$ and define a set of arrows A between those classes. An arrow $(\mathcal{F}^i, \mathcal{F}^j)$ indicates that fuzzers of class $\mathcal{F}^i$

function as explorers for fuzzers in the class F^j, i.e. fuzzers in F^i attract fuzzers in $\mathcal{F}^j$ according to the guidance algorithm as depicted in Figure 7.1. This yields the directed graph

$$\mathcal{G} = \left(\bigcup_{i \in \{1,...,n_K\}} \mathcal{F}^i,\ A \right). \tag{7.7}$$

For example, setting $n_K = 2$ yields the previously described situation of a single colony of explorers $\dot{\mathcal{F}} = \cup_{i \in \{1,...,n_E\}} \dot{F}^i$ attracting a single colony of workers $\mathcal{F} = \cup_{j \in \{1,...,n_W\}} F^j$, i.e.

$$\mathcal{G}_2 = \left(\dot{\mathcal{F}} \cup \mathcal{F}, (\dot{\mathcal{F}}, \mathcal{F}) \right). \tag{7.8}$$

Such a graph definition is especially useful when distributing fuzzing instances on actual computer clusters. If the fuzzing campaign is executed by a heterogenous hardware infrastructure the graph should be adapted according to bandwidth and latency of the respective interconnection network. In particular, if two fuzzing classes in the graph are connected with an arrow, they are suited to be placed in the same high bandwidth and low latency interconnection network. Vice versa, if two fuzzer classes have a large geodesic distance in the graph (i.e. a relatively high number of arrows in the shortest path between them), they may be distributed accordingly in computing clusters that are interconnected with lower bandwidth and higher latency.

7.2.5. Choices for f and g

In choosing the spraying function f and the attraction function g we are guided by the following considerations. f determines the actual attracting fraction of the explorer traces. As determined by Equation (7.2) a fast decay of f leads to short attracting traces and vice versa. In the extreme, f distributes the attractant nowhere on the explorer trace except on the most recently computed center (corresponding to $f(0) \neq 0$ and $f(t) = 0$ for all $t > 0$). For simultaneous strong force of attraction such a choice is almost equivalent to direct corpus synchronization. However, we want the black-box workers F^j $(j = 1, ..., n_W)$ to be guided along the explorer paths for two reasons: Close proximity to actually all regions roamed by the explorers and enough time for black-box worker exploration. To be more precise, for large periods t' of pure fuzzing (as indicated in Figure 7.1) corpus extraction provides only discrete snapshots of current explorer positions in time. During fuzzing for time t' the workers also diffuse their corpora through input space. Too high attractivity of the most currently generated explorer corpus would tend to ignore fuzzing the whole path between extracted corpus snapshots. Since we want the black-box workers to follow the explorer paths as closely as possible while simultaneously give them enough time to generate corner cases not discovered by the explorers, we distribute the mass of f accordingly. As shown in our evaluation in Section 7.3 we achieved good results with different Gaussian functions for f. Regarding the attracting function g in Equation (7.6) we borrow from the law of gravity and propose higher attraction forces for higher concentrations and closer distances. We implement a sigmoid function made of two logistic functions in Section 7.3.

7.3. Implementation and Evaluation

To show the feasibility of our approach we implemented a prototype of the algorithm as depicted in Figure 7.1 with one dedicated group of explorers guiding a colony of high throughput worker fuzzers. In this section we first present our choices for functions f and g, and subsequently evaluate our method.

For spraying function f we implemented Gaussian functions

$$f(t) = \begin{cases} c_1 e^{-\frac{(t-c_2)^2}{2c_3^2}} & 0 \leq t < t_z \\ 0 & t \geq t_z \end{cases} \tag{7.9}$$

parameterized by $c_1, c_2, c_3 \in \mathbb{R}_{>0}$. While c_1 determines the total amount of attractant, c_3 controls the decomposition rate of attractant concentration on the traces T^i. A nonzero value of $c_2 > 0$ translates to an attractant that unfolds its full attractive potential only with a time delay, but we set $c_2 = 0$ for the following benchmarks.

Function g assigns the force of attraction dependent on attractant concentration and distance to the attractant. Shorter distance and higher concentration should result in stronger attraction. We implemented $g : \mathbb{R}^2_{\geq 0} \to [0, 1]$ as

$$g(f, \delta) = \Big(\Big(1 + a_1 e^{a_3 - a_2 f}\Big) \Big(1 + d_1 e^{d_2 \delta - d_3}\Big) \Big)^{-1}, \tag{7.10}$$

where $(f, \delta) = \big(f(t - \tau_i), \delta(\hat{c}^i, x)\big)$ denote attractant concentration and distance, respectively, and $a_1, a_2, a_3 \in \mathbb{R}_{>0}$.

As testing target we chose the command line tool `djpg` for decompressing `JPEG` files to image files (in `BMP` and `GIF` format). All explorers are slow moving versions of the fuzzer presented in Chapter 5. We initialized both the explorers and black-box workers with seed corpora containing image files of size 100 kB. Then we measured the distance δ between most recently generated centers (corresponding to the end of trace $\bar{T}$) of a selected explorer and the respective corpus centers of a successfully attracted black-box worker.

Figure 7.2 depicts attraction behavior of a single explorer ($n_E = 1$). After each of the first 100 iterations of the *do-while* loop of our algorithm (as depicted in Figure 7.1) we indicate distance δ on the z-axis. After 100 iterations we stop the fuzzing campaign and increase the force of attraction by increasing d_2. After 10 fuzzing campaigns ($d_2 = 4, ..., 14$) we receive the surface depicted in Figure 7.2. For strong forces of attraction (corresponding to high values of d_2) the single explorer successfully attracts all workers and reduces the mean distance δ from averaged 400 kbit to 330 bit. Weak forces of attraction (corresponding to low values of d_2) do not lead to attraction. This is due to the diffusivity of worker corpora in input space: With a black-box fuzzer mutation rate of $r = 4 * 10^{-5}$ the workers diffuse their test case corpora stronger than the explorer attracts them.

In a second experiment we increase the number of explorers to $n_E = 5$ as well as the black-box mutation ratio to $r = 8 * 10^{-4}$. The resulting benchmark is depicted in

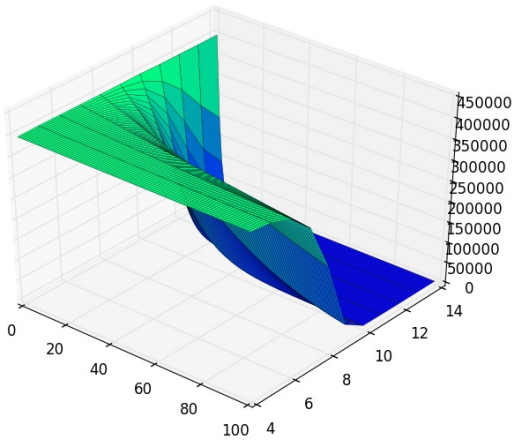


Figure 7.2.: Attraction of a single explorer ($n_E = 1$) within the first 100 iterations, resulting in a decrease of δ from averaged 400 kbit down to 330 bit, where mutation ratio for the measured black-box fuzzer is $r = 4 * 10^{-5}$. Increasing d_2 from 4 to 14 causes a significant stronger attraction.

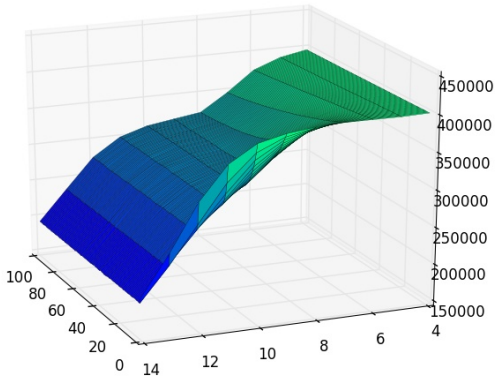


Figure 7.3.: Attraction for $n_E = 5$ within the first 100 iterations and increasing $d_2 = 4, ..., 14$, causing a decrease of δ from averaged 400 kbit down to 190 kbit, where mutation ratio is $r = 8 * 10^{-4}$.

Figure 7.3. Analog to the previous setting we measure distance δ (on the z-axis) in each of the first 100 iterations for 10 fuzzing campaigns with respectively increasing force of attraction $d_2 = 4, ..., 14$. After successful initial attraction the distance δ reaches an equilibrium state depending on the force of attraction. We can successfully reduce the distance and guarantee proximity of the workers to the explorer traces by increasing the force of attraction. The equilibrium of distance between an attracted worker and its explorer guide is caused by three antagonizing forces: Attraction by its guide, attraction by all competing explorers, and mutation ratio of the worker. Increasing the force of attraction simultaneously for all explorers binds the worker further to its guide (because of the sigmoid form of g as defined in Equation (7.10)) and additionally overcomes even high mutation ratios.

7.4. Discussion

In this section we discuss characteristics, possible modifications, and expansions of our approach.

As shown in our evaluation once a black-box worker fuzzer has joined an explorer it will remain there most probably for the rest of the fuzzing campaign. However, the attraction mechanism of the group of explorers after each period of time t' brings in a small fraction of valuable fresh input from the surrounding explorers. This is due to the construction of our attraction mechanism as described in Section 7.2.2, where actual attraction is lowering the Hamming distance by flipping bits to match the attracting center (which is the test case that matches the majority of test cases of a current explorer corpus regarding the bit string). Therefore, we achieve mixing of test cases between essentially isolated explorers.

Further, attraction of the trace of an explorer as sprayed by f according to Equation (7.2) guarantees optimal post-processing of input regions toughed by the explorers. As discussed in Section 7.2 trace attraction gives two vital advantages compared to simple corpus synchronization: Close proximity to actually all regions roamed by the explorers and enough time for black-box worker exploration. Since black-box workers are significantly faster and provide different mutation engines, their concentration around explorer traces often reveals new side paths and corner cases that the explorers did not discover.

We put much emphasis on out-of-the-box deployment of existing fuzzing frameworks to avoid any possibly time-consuming or (in case of closed source fuzzers) impossible modifications. However, access to information inside the explorer fuzzers would allow us to adapt attraction behavior for each explorer individually. Our presented spraying mechanism as determined by function f in Equation (7.2) treats each explorer equally: It assumes the explorer has found a region of quality test cases (e.g. regarding code coverage), sprays the corpus center, and lets the concentration descent over time. If an explorer discovers significantly more new basic blocks than all other explorers, we should be able to assign a higher force of attraction to respective test cases. In other words, comparing the numbers of newly discovered basic blocks found by the individual explorers would allow us to allocate higher attractant concentrations to centers of higher

quality corpora, enabling strongest attraction to the currently best performing explorer. Further, we could introduce a repellent inducing negative chemotaxis for test cases that for example consume too much time to process or enter code regions that are not relevant for testing.

So far we do not provide any feedback from the black-box fuzzers back to the explorers. This is motivated by the nature of basically blind black-box fuzzers which do not obtain any information from the targeted binary during runtime, except a program crash. But especially this crash information could be used to mark the corresponding test case as attractive. Such modifications could improve the overall fuzzing campaign.

7.5. Conclusion

Inspired by insect and animal colonies that reveal a rich diversity of scouts and explorers we introduce the first framework for large-scale random testing of binary executables based on the concept of chemotaxis. In order to maximize coverage of execution paths in the target under test we draw fast and efficient (but blind regarding runtime information) black-box workers to regions in input space discovered by feedback-driven explorers. We realize this by constructing a mechanism for distributing attractants in input space and defining the resulting force field of attraction exerted on black-box fuzzers. This approach combines the best of both worlds: The sight into runtime information from dynamic instrumentation by the explorers and the speed of black-box worker fuzzers. Next, we generalize this approach to multiple hierarchies of fuzzers to capture their attraction network in a graph. Such a graph definition is especially useful when distributing fuzzing instances on actual computing clusters, as we can adjust the graph of attraction to the hardware infrastructure. We show the feasibility of our approach by evaluating it on a real-world target with different parameter settings. Further, we discuss modifications and expansions of our algorithm. Especially customized testing frameworks would allow us to distribute attractant concentration significantly more fine-grained, which probably results in faster code coverage and is subject to future work.

Part III.

Fuzzing with Learning Behavior

We finally enter the third and last phase of our journey. Let us take the bird's eye view first. Our mathematical model of Part I enabled us to translate specific stochastic processes into algorithms for software testing in Part II. The model spans a rich world that invites for exploration with different techniques and search strategies. Up to now, we only explored a small part of this world as we did not take multiple action classes or actual state structures into account. If we do so, we need to enhance our equipment and means of transportation. How can we explore the full range of our model? In this chapter we approach this question with some help from the field of machine learning.

Considering the full capabilities of our model, we soon realize the complexity we have to deal with. In the reduced setting of similar actions and rewards independent of states we were able to find adequate fuzzing strategies: We got inspired by biology in choosing processes with suitable characteristics which directly determine fuzzing behavior (as formalized in Equation 3.10 on page 25). For example, while the algorithm introduced in Chapter 5 self-adapts its diffusivity and corresponding actions according to the received reward, it always performs a Lévy flight in nature. From the bird's eye view, we constructed a subset of actions with known similar characteristics within $\mathcal{A}$ (as defined in Equation 4.4) based on stochastic processes and defined a reward-based decision mechanism for the actions to take (in terms of Equation 4.5 on page 32).

Let us enlarge our problem space by (1) offering the system multiple different classes of actions and (2) taking the system state into account. In this setting two questions arise:

- Do optimal behaviors $(\pi_t)_{t\in\mathbb{N}} \in \Pi$ exist?
- Is the definition of such a behavior accessible for a human?

For discrete and finite state and action spaces, Watkins [116] theoretically provides a positive answer to the first question. However, our state space (as defined in Equation 4.2 on page 30) is huge and our action-value function $Q(x, a)$ cannot be represented in a look-up table. Therefore, we are in a similar situation as in chess or the game of Go: While an optimal strategy might exist, we are currently bound to playing the game in the search for it. In the same spirit, while we currently cannot give a satisfactory answer to the first question, we can play the game against the program. In fact, we can apply the same techniques that achieved super-human behavior in Backgammon [5, 6], Atari games [7], and the game of Go [8]. We can do so because the mathematical model of Part I provides a direct interface to reinforcement learning. And similar to gaming, we expect that the definition of winning behaviors for fuzzing are not accessible for the human, especially if we take into account states from binary input. In the following we will see that deep Q learning turns out to be a fruitful new direction in software testing.

8. Reinforcement Fuzzing

"But an effect can become a cause, reinforcing the original cause and producing the same effect in an intensified form, and so on indefinitely."

George Orwell

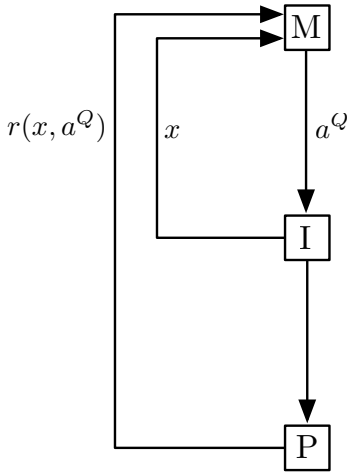


Figure 8.1.: Reinforcement Fuzzing.

In this part we formulate random test generation for fuzzing as a reinforcement learning problem. Modeling basic characteristics of random test generation as a Markov decision process as described in detail in Part I of this work enables us to apply state-of-the-art deep Q-learning algorithms that optimize predefined rewards measured during runtime of the program under test. By observing the reward effects caused by mutating with a set of actions performed on the seed file, the fuzzing agent learns a policy that converges to the optimal behavior within the defined setting. To indicate the feasibility of our approach we implement and evaluate a prototype of such a reinforcement learning agent. We experiment with two different types of rewards to show that our fuzzing algorithm learns to select highly rewarded string mutation actions better than a baseline of purely random strategies.

8.1. Motivation

Our mathematical model presented in Part I of this work theoretically captures rewards that directly take the system state as defined in Equation 4.5 on page 32 into account. Further, instead of a single class of similar actions, our model permits a generic actions space $\mathcal{A}$ (as defined in Equation 4.4) that covers all possible string mutations. As discussed at the beginning of this part, in the following we will explore the full range of our model: Let us enlarge our problem space by (1) offering the system multiple different classes of actions and (2) taking the system state into account. In this setting two questions immediately arise:

We propose an effective fuzzing method within the given problem space by formulating random test generation for fuzzing as a reinforcement learning problem. The reinforcement learning setting defines an agent that interacts with a system. Each performed action causes a state transition of the system. Upon each performed action the agent observes the next state and receives a reward. The goal of the agent is to maximize the total reward over time. We introduced this setting in Part I of this work. In particular, we experiment with Q-learning, that just recently has successfully applied similarly complex scenarios [8, 7, 5, 6].

In summary, we make the following contributions:

- We formulate fuzzing as a reinforcement learning problem.
- We introduce a fuzzing algorithm based on deep Q-learning that learns to choose highly rewarded actions given an observed input string.
- We implement and evaluate a prototype of our defined approach.

The remainder of this paper is organized as follows. In Section 8.2 we discuss related work. We introduce our reinforcement fuzzing algorithm in Section 8.4. Next, we give details regarding our implementation and evaluation in Section 8.5 and discuss properties and possible expansions of our algorithm in Section 8.6. We conclude with a short outlook in Section 8.7.

8.2. Related Work

Our approach for reinforcement fuzzing in this chapter is influenced by two main streams of research: Grammar reconstruction and reinforcement learning.

Our fuzzing algorithm we introduce in this chapter initially does not have any specification or grammar regarding the input format and therefore initially acts like a mutation fuzzer. However, during the fuzzing process it learns to perform optimally rewarded input mutations based on the observed byte string. Performing actions (like token insertion, token deletion, or random bit mutation) specific to an observed input strings can be interpreted as a generalized grammar. Research on constructing grammars for generation fuzzing made significant progress from its beginnings in the early 1970's [117, 118] until recent achievements [104, 87, 84, 119]. The authors of [120] propose an algorithm for

automatic synthesis of a context-free grammar given a set of seed inputs and a black-box target. Cui et al. [61] can automatically detect record sequences and types in the input by identification of chunks based on taint tracking input data in respective subroutine calls. Similarly, the authors of [62] apply dynamic tainting to identify failure-relevant inputs. Another recent approach was proposed by Höschele et al. [63], who mine input grammars from valid inputs based on feedback from dynamic instrumentation of the target by tracking input characters. In contrast, our fuzzer creates a generalized grammar over time that is especially necessary when dealing with binary input. We discuss this aspect further in Section 8.6.

Research on reinforcement learning [59] emerged from trial and error learning, and optimal control for dynamic programming [121]. Especially the Q learning approach introduced by Watkins [122, 116] was recently combined with deep neural networks [7] to achieve impressive results in complex tasks like playing the game of Go against human. We apply deep Q networks to learn fuzzing policies that perform optimally rewarded actions in the face of a given input.

Combining machine learning with fuzzing is a novel approach and to the best of our knowledge there is just one single effort into this direction: The authors of [123] apply neural networks to generate fuzzing inputs with a sequence to sequence algorithm.

8.3. Q-Learning

In this section we give the necessary background on Q-learning and motivate the application of deep Q-networks.

As discussed in Chapter 3 in Part I of this work, for each state-action pair $(x, a) \in X \times A$ and each $U \subset X \times \mathbb{R}$ the kernel P_0 gives the probability $P_0(U|x, a)$ that performing action a in state x causes the system to transition into some state of X and yielding some real value reward as indicated by U. P_0 directly provides the state transition probability kernel P for single transitions $(x, a, y) \in X \times A \times X$

$$P(x, a, y) = P_0(\{y\} \times \mathbb{R}|x, a). \tag{8.1}$$

This naturally gives rise to a stochastic process: An agent observing a certain state chooses an action to cause a state transition with corresponding reward. By subsequently observing state transitions with corresponding rewards the agent aims to learn an optimal behavior that earns the maximal possible cumulative reward over time. Formally, with the stochastic variables $(y(x, a), R(x, a))$ distributed according to $P_0(\cdot|x, a)$ the expected immediate reward for each choice of action is given by $\mathbb{E}[R(x, a)]$. During the stochastic process $(x_{t+t}, R_{t+1}) \sim P(\cdot|x_t, a_t)$ the aim of an agent is to maximize the total discounted sum of rewards

$$\mathcal{R} = \sum_{t=0}^{\infty} \gamma^t R_{t+t}, \tag{8.2}$$

where $\gamma \in (0, 1)$ indicates a discount factor that prioritizes rewards in the near future. The choice of action a_t an agent makes in reaction to observing state x_t is determined

by its policy $a_t \sim \pi(\cdot|x_t)$. Let

$$Q^{\pi}(x, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | x_0 = x, a_0 = a\right] \tag{8.3}$$

denote the expected cumulative reward for an agent that behaves according to policy π. Then we can reduce our problem of approximating the best policy to approximating the optimal Q function. One practical way to achieve this is adjusting Q after each received reward according to

$$Q(x_t, a_t) \leftarrow Q(x_t, a_t) + \alpha \left(R_t + \gamma \max_a Q(x_{t+1}, a) - Q(x_t, a_t)\right), \tag{8.4}$$

where $\alpha \in (0, 1]$ indicates the learning rate. The process in this setting works as follows: The agent observes a state x_t, performs the action

$$a_t = \arg\max_a Q(x_t, a) \tag{8.5}$$

that maximizes the total expected future reward and thereby causes a state transition from x_t to x_{t+1}. Receiving reward R_t and observing x_{t+1} the agent then considers the best possible action $\arg\max_{a_{t+1}} Q(x_{t+1}, a_{t+1})$. Based on this consideration, the agent updates the value $Q(x_t, a_t)$. If for example the decision of taking action a_t in state x_t led to a state x_{t+1} that allows to choose a high reward action and additionally invoked a high reward R_t, the Q value for this decision is adapted accordingly. Here, the factor α determines the rate of this Q function update.

For small state and action spaces, Q can be represented as a table. However, for large state spaces we have to approximate Q with an appropriate function. An approximation using deep neural networks was introduced just recently by the authors of [7]. For such a representation the update rule in Equation (8.4) directly translates to minimizing the loss function

$$L = \left(r + \gamma \max_a Q(x_{t+1}, a) - Q(x_t, a_t)\right)^2. \tag{8.6}$$

The learning rate α in Equation (8.4) then corresponds to the rate of stochastic gradient descent during backpropagation.

Deep Q-networks have been shown to handle large state spaces efficiently. This allows us to define an end-to-end algorithm directly on raw input strings, as we will see in the next section.

8.4. Reinforcement Fuzzing Algorithm

In this section, we present the overall reinforcement fuzzing algorithm. In our approach we use Markov decision processes to generate test cases and adapt their behavior according to feedback from dynamic instrumentation of the target.

8.4.1. Initialization

We start with an initial seed $x \in \mathcal{I}$. Theoretically, the choice of x is not constrained in any way, it may not even be valid with regard to the input format of the target. However, in our evaluation in Section 8.5 we show that correctly formated seeds yield better results. Further, we initialize the Q function. Practically, we apply a deep neural net that maps states to the estimated Q values of each action, i.e. we simultaneously approximate the Q values for all actions A given a state $x' \in S(x)$ as defined in Equation (4.1). The $x' \mapsto Q(x', a)$ representation provides the advantage that we only need one forward pass to simultaneously receive the Q values for all actions $a \in A$ instead of $|A|$ forward passes. During Q function initialization we distribute the network weights randomly.

8.4.2. State Extraction

The state extraction step takes as input a seed $x \in \mathcal{I}$ and outputs a substring of $x' \in S(x)$. In Chapter 4 of Part I of this work we defined the states of our Markov decision process to include $\mathcal{I} = \Sigma^*$. For the given seed $x \in \mathcal{I}$ we extract a strict substring $x' \in S(x)$ at offset $o \in \{0, ..., |x| - |x'|\}$ of width $|x'|$. In words, the seed s corresponds to the system and the reinforcement agent observes a fragment of the whole system via the substring x'. We experimented with controllable (via action) and predefined choices of offsets and substring widths, as discussed in Section 8.5.

8.4.3. Action Selection

The action selection step takes as input the current Q function representation and an observed state x' and outputs an action $a \in A$ as defined in Equation (4.4). Actions are selected according to the policy π following an ϵ-greedy behavior: With probability $1 - \epsilon$ (for a small $\epsilon > 0$) the agent selects an action

$$a = \arg\max_{a'} Q(x', a') \tag{8.7}$$

that is currently estimated optimal by the Q-function, i.e. it exploits the best possible choice based on experience. With a probability ϵ it explores any other action, where the probability of choice is uniformly distributed within $|A|$.

8.4.4. Mutation

The mutation step takes as input a seed x and an action a. It outputs the string that is generated by applying action a on x. As indicated in Equation (4.4) we define actions to be mappings to probabilistic rewriting rules and not rewriting rules on their own. So applying action a on x means that we mutate x according to the rewrite rule mapped by a within the probability space $(\mathcal{I} \times \mathcal{I},\ \mathcal{F}, P)$. We make this separation to distinct between the random nature of choice for the action $a \sim \pi(\cdot|x)$ and the randomness within the rewrite rule. For example, in Section 8.5 we define rewrite rules to be random bit flips according to a predefined mutation ratio.

8.4.5. Reward Evaluation

The reward evaluation step takes as input the target program P, an action $a \in A$, and an input $x \in \mathcal{I}$ that was generated by the application of a on a seed. It outputs a positive number $r \in \mathbb{R}^+$. As indicated in Equation (4.5) the stochastic reward variable $R(x, a) = E(x) + G(a)$ sums up the rewards for both generated input and selected action. Function E rewards characteristics recorded during target program execution, such as the number $E_1(x, I')$ of newly discovered basic blocks given a history of previously generated inputs $I' \subset I$ as defined in Equation (4.6), the total coverage in terms of unique basic blocks $E_1(s, \{\})$ without mutation history, or the time it takes the target program to process x. According to the search heuristics, the reward should depend monotonically increasing on coverage numbers. In contrast, actions with higher computational complexity should be rewarded less. With such a definition of R the Q-learning algorithm is reinforced to find an equilibrium between coverage advancements and action processing costs.

8.4.6. Q-Update

The Q-update step takes as input the extracted substring $x' \in S(x)$, the action a that generated x, the evaluated reward $r \in \mathbb{R}^+$, and the Q function approximation, which in our case is a deep neural network. It outputs the updated Q approximation. As indicated above, the choice of applying a deep neural network Q is motivated by the requirement to learn on raw substrings $x' \in S(x)$. The Q function predicts for a given state the expected rewards for all defined actions of A simultaneously, i.e. it maps substrings according to $x' \mapsto Q(x', a)$. We update Q in the sense that we adapt the predicted reward value $Q(x_t, a_t)$ according to the target $r + \gamma \max_a Q(x_{t+1}, a)$. This yields the loss function L given by Equation (8.6) for action a_t. All other actions $A \setminus \{a_t\}$ are updated with zero loss. The convergence rate of Q is primarily determined by the learning rate of stochastic gradient descent during backpropagation as well as the choice of γ.

8.4.7. Joining the Pieces

Now that we have presented all individual steps we can proceed with combining them to the overall fuzzing algorithm as depicted in Figure 8.2.

We start with an initialization phase that outputs a seed x as well as the initial version of Q. Then, the fuzzer enters the loop of state extraction, action selection, input mutation, reward evaluation, Q update, and test case reset. Starting with a seed $x \in \mathcal{I}$ the fuzzer extracts a substring $x' \in S(x)$ and based on the observed state x' the chooses the next action according to its policy. The choice is made looking at the best possible reward predicted via $x' \mapsto Q(x', a)$ and applying an ϵ-greedy exploitation-exploration strategy. To guarantee initial exploration we initially define a relatively high value for ϵ and monotonically decrease ϵ over time until it reaches a final small threshold, from then on it remains constant. The selected action provides a string substitution as indicated in Equation (4.4) which is applied to x for mutation. The generated mutant is feed into the target program P to evaluate the reward r. Together with Q, x, and a, this reward is taken into account for Q update. Finally, we periodically reset input x to a valid seed.

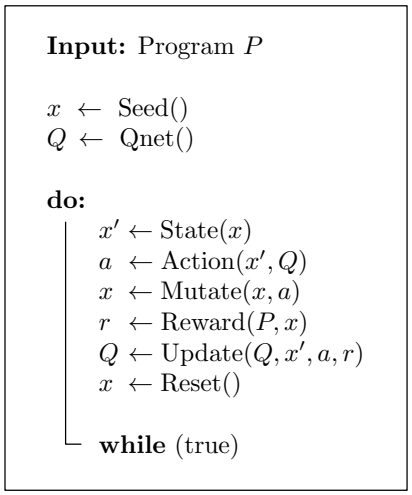


Figure 8.2.: Reinforcement fuzzing algorithm.

This is motivated by the observation that mutating valid input resulted in significantly better results, as described in Section 8.5. After reset the fuzzer continues the loop.

The described algorithm describes reinforcement fuzzing with activated policy learning. We show in our evaluation in Section 8.5 that the Q-network generalizes on states. This allows us to switch to high-throughput mutant generation with a fixed policy after a sufficiently long training phase.

8.5. Implementation and Evaluation

To indicate the feasibility of our approach we implemented a prototype of the reinforcement fuzzing algorithm as depicted in 8.2. In this section we present details regarding our implementation together with an evaluation of the prototype.

8.5.1. Target

As fuzzing target we chose programs processing files in the Portable Document Format PDF. This format is complex enough to provide a realistic testbed for evaluation. From the 1300 pages long PDF specification we just need the following basic understanding: Each PDF document is a sequence of PDF bodies each of which includes three sections named objects, cross-reference table, and trailer. Simplified, the trailer at the end of a PDF file points to the beginning of the cross-reference table, which lists the offsets of all objects in the object section, which in turn makes up by far the largest part of a file. While our algorithm is defined to be independent of the targeted format, we make use of this structure to define actions specifically crafted for PDF object. Therefore, we parse the cross-reference table to get the offsets of all objects within the seed file.

We considered different PDF processing programs including the PDF parser in the Microsoft Edge browser on Windows and several command line converters on Linux. All

results in the following presentation refer to fuzzing the *pdftotext* program mutating a 168 kByte seed file with 101 PDF objects including binary fields.

8.5.2. Implementation

In the following we present details regarding our implementation of the proposed reinforcement fuzzing algorithm. We apply existing frameworks for binary instrumentation and neural network training and implement the core framework including the Q-learning module in Python 3.5.

State Implementation Our fuzzer observes and mutates input files represented as binary strings. With $\Sigma = \{0, 1\}$ we can choose between state representations of different granularity, for example bit or byte representations. We encode the state of a substring x' as the sequence of bytes of this string. Each byte is converted to its corresponding float value when processed by the Q network. As introduced in Section 8.4 we denote $o \in \{0, ..., |x| - |x'|\}$ to be the offset of x' and $w = |x'|$ to be the width of the current state.

Action Implementation We implement each action as a function in a Python dictionary. As string rewriting rules we take both probabilistic and deterministic actions into account. In the following we list the action classes we experiment with.

- *Random Bit Flips.* There are two types of actions that perform random bit flips. Each class mutates the substring x' with a mutation ratio. Actions of the first type mutate x' with a predefined fixed mutation ratio. We defined four actions to mutate with fixed ratios. Actions of the second type adjust a global mutation ratio and then flip the bits of x' according to this ratio. We defined three actions of this type: One that increases the global ratio, one that leaves the ratio constant, and one that decreases the ratio. All actions of the second type mutate x' after ratio adjustment.
- *Insert Dictionary Tokens.* We define a single action that inserts tokens from a predefined dictionary. The tokens in the dictionary consist of ASCII strings extracted from a set of selected seed files.
- *Shift Offset and Width.* We define four actions that shift the offset and width of the observed substring. Left and right shift take place at the PDF object level. Increasing and decreasing the width take place with byte granularity.
- *Shuffle.* We define two actions for shuffling substrings. The first action shuffles bytes within x', the second action shuffles three segments of the PDF object that is located around offset o.
- *Copy Window.* We define two actions that copy x' to a random offset within x. The first action inserts the bytes of x', the second overwrites bytes.
- *Delete Window.* We define an action that deletes the observed substring.

Reward Implementation For evaluation of the reward $R(x, a) = E(x) + G(a)$ we experimented with both coverage and execution time information. We did not take into account a reward depending on the selected action. If we expand the action space to include symbolic execution, taint-based analysis, or seq2seq algorithms, we could introduce a negative reward for the time it takes to perform the action. However, this is subject to future work.

To measure $E(x) = E_1(x, \mathcal{I}')$ as defined in Equation (4.6) we made use of existing instrumentation frameworks. We initially experimented with Microsoft iDNA and NIRVANA for measurements involving the PDF parser included in Edge. The iDNA setting recorded each machine instruction of each loaded DLL during target program execution. However, to speed up training of the Q net we switched to smaller parser targets: On Linux we implemented a custom Intel PIN tool that counts the number of unique basic blocks within the *pdftotext* program.

Q Network Implementation We implemented Q in the Google Tensorflow framework by constructing a feed forward neural network with four layers connected with nonlinear activation functions. We initialize the weights randomly and uniformly distributed within $w_i \in [0,\ 0.1]$. The initial learning rate of the gradient descent optimizer is set to 0.02. From all activation functions provided by the Tensorflow framework, we found the *tanh* function to yield the best results for our setting.

8.5.3. Evaluation

In this section we evaluate our implemented prototype. We present improvements to a predefined baseline and also discuss current limitations. All measurements were performed on a Xeon E5-2690 2.6 Ghz with 112 GB of RAM.

Baseline

To show that the defined algorithm actually learns to perform high rewarded actions given an input observation we measure the baseline in terms of rewards for a policy that randomly selects actions, where the choice is uniformly distributed among the action space A. While this is quite a low baseline in terms of overall fuzzing performance, it serves as an adequate reference point in terms of policy learning rate. Formally, actions in the baseline policy π_B are distributed uniformly according to $a \sim \pi_B(\cdot|x)$ and $\forall a \in A :\ \pi_B(a|x) = |A|^{-1}$. After $n_g = 1000$ generations we calculated the quotient of the most recent 500 accumulated rewards by our algorithm and the baseline to measure the relative improvement.

Replay Memory

We experimented with two types of agent memory: The recorded state-action-reward-state sequences as well as the history of previously discovered basic blocks. Our algorithm did not show any improvement compared to the baseline with these two types of memories.

The first type of memory is established during the fuzzing process by storing sequences $e_t := (x_t, a_t, r_t, x_{t+1})$ in order to regularly replay samples of them in the Q-update step. For each replay step at time t a random experience out of $\{e_1, ..., e_t\}$ is sampled to train the Q network. As discussed in [7] experience replay should provide several advantages, e.g. countering strong correlations between trailing samples.

As indicated in Equation 4.6 we defined the coverage evaluation $E_1(x, I')$ to take a history of basic blocks into account that have been previously discovered by executing inputs in I'. Initially, we kept all previously visited basic blocks in that history to reward only new discoveries. This did not result in any improvement with regard to the baseline. Then, we implemented I' to include only the $n_H \in \mathbb{N}$ previously executed test cases. This short term memory also did not increase total rewards with respect to the baseline. Only a memoryless choice of $I' = \emptyset$ yielded good results. This behavior is explained by the sparsely received rewards over time: After a while of fuzzing the discovery of new basic blocks becomes rare and therefore the rewards become sparse. Learning with sparsely distributed rewards is an active area of research and not yet fully understood [124]. Therefore, in the following we set $I' = \emptyset$ so that $E_1(x, \{\})$ indicates execution path length of the target given an input x. Regarding our algorithm as depicted in Figure 8.2 we reset the basic block history after each step via the *Reset*() function. Since experience replay of state-action-reward-state sequences requires a history of previously discovered basic blocks, we similarly did not achieve any improvement in comparison to the baseline.

Since both types of agent memory did not yield any improvement in comparison to the baseline, we switched them off for the following measurements. Further, we deactivated all actions that do not mutate the seed input, e.g. random bit flip actions of the second type (altering the global mutation ratio) or shifting offsets and state widths. Instead of active offset o and state width $w = |x'|$ selection via an agent action, we set the offset for each iteration randomly, where the choice is uniformly distributed within $\{0, ..., |x| - |x'|\}$ and fixed $w = 32$ Bytes.

Choices of Rewards

We experimented with three different types of rewards: Maximization of code coverage $R_1(x, a) = E_1(x, \{\})$, execution time $R_2(x, a) = E_2(x) = T(x)$, and a combined reward $R_3(x, a) = E_1(x, \{\}) + T(x)$ for multi-goal fuzzing. While $R_1(x, a)$ is deterministic, $R_2(x, a)$ comes with minor noise in the time measurement. Measuring the execution time for different seeds and mutations revealed a variance that is two orders of magnitude smaller than the respective mean so that R_2 is stable enough to serve as a reliable reward function. All three choices provided improvements with respect to the baseline. For example, Figure 8.3 depicts a direct reward comparison between reinforced fuzzing and the defined baseline with time reward R_2 for the first 1000 generations. We apply Savitzky-Golay filters to indicate a median. Our proposed fuzzing algorithm cumulates in average 7% higher execution time reward in comparison to the baseline.

In comparison rewarding coverage according to $R_1(x, a)$ yielded rewards as depicted in Figure 8.4. Rewarding coverage with $R_1(x, a)$ provided slower increasing learning improvement compared to the baseline as a function of input generation steps. Further, it

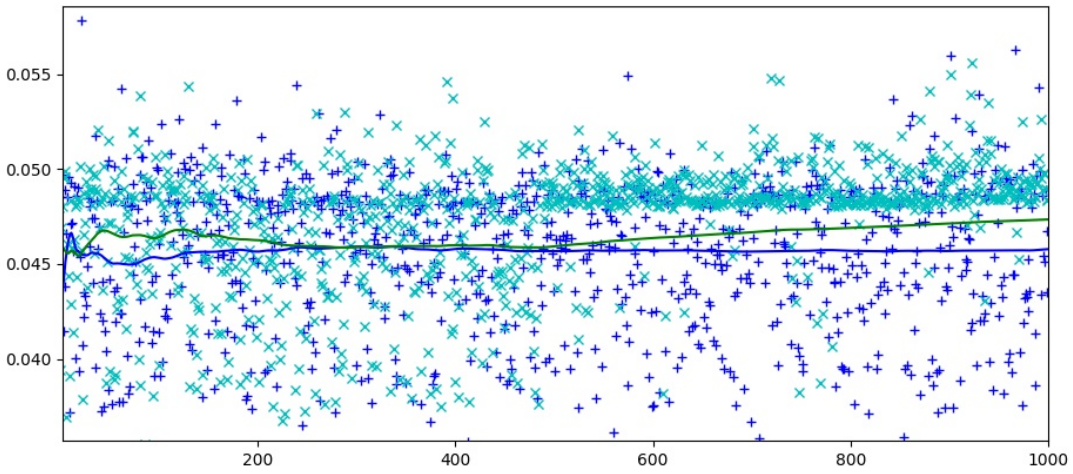


Figure 8.3.: $R_2 - n_g$ diagram for $w = 32$, $\epsilon = 0.1$, and $\gamma = 0.2$ for $Q-$learning (cyan) and baseline (blue) rewards.

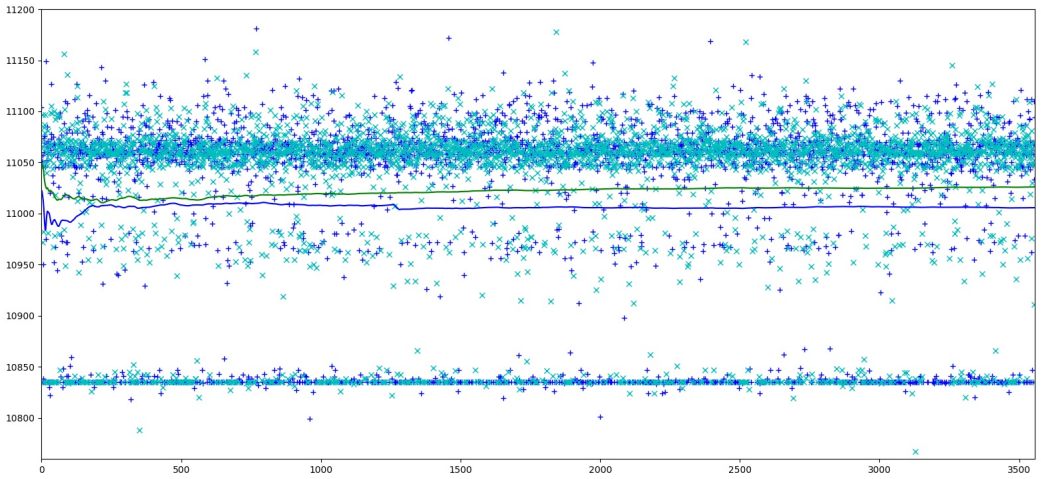


Figure 8.4.: $R_1 - n_g$ diagram for $w = 32$, $\epsilon = 0.1$, and $\gamma = 0.2$ for $Q-$learning (cyan) and baseline (blue) rewards.

was significantly slower than execution time rewards due to the high overhead of binary instrumentation. Further, the average reward of reinforcement fuzzing is not a monotonically increasing function of the learning iterations: Figure 8.3 clearly shows a decay of average reward between generations 130 and 440. This corresponds to explorative behavior of the learning algorithm. As stated in Section 8.4 we implemented an ϵ-greedy exploitation-exploration strategy. To guarantee initial exploration we monotonically decrease ϵ over time until it reaches a final small threshold ϵ_0 according to

$$\epsilon(i) = \begin{cases} \left(\frac{i}{\epsilon_1} + 1\right)^{-1} & i < i_0 \\ \epsilon_0 & i \geq i_0 \end{cases} \tag{8.8}$$

where i indicates the number of learning iterations, i.e. number of executed loops of the main fuzzing algorithm as depicted in Figure 8.2.

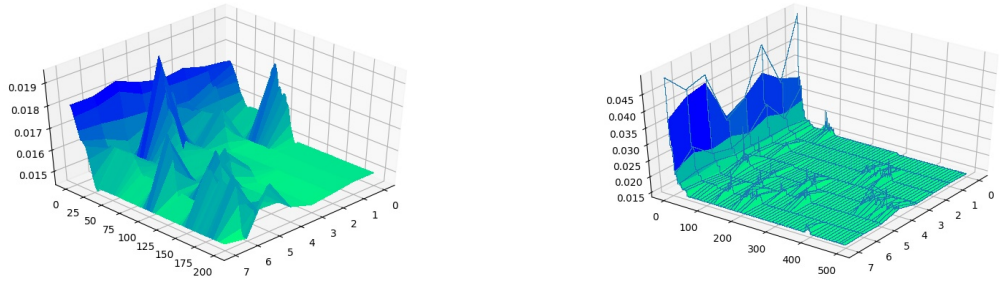
Interestingly, coverage measurement as depicted in Figure 8.4 indicates certain levels of execution path depths that could not be observed by rewarding execution time as depicted in Figure 8.3. Since both time and coverage rewards yielded improvements in comparison to the baseline, the question arises to what extend those two types of rewards correlate: We measured an average Pearson correlation coefficient of 0.48 between coverage R_1 and execution time R_2. This correlation motivates the combined reward $R_3(x, a) = E_1(x, \{\}) + T(x)$, where $T(x)$ is a simple rescaling of execution time by a multiplicative factor $1 * 10^6$ so that the execution time contributes to the reward equitable to E_1. Training the Q net with R_3 yielded an improvement of 11.3% in execution time. Note that this result is better that taking exclusively R_1 or R_2 into account. There are two explanations standing to reason for this result: First, the noise of time measurement could introduce rewarding explorative behavior of the Q net. Considering that the variance of execution time is two orders of magnitude smaller than the respective mean and that we already implemented an ϵ-greedy policy for exploration, this explanation seems unlikely. Second, deterministic coverage information could add stability to R_2.

State Width

Increasing the state width $w = |x'|$ from 30 Bytes to 80 Bytes decreased the improvement (measured in average reward $R_2(x, a)$ compared to the baseline) from 7% to 3.1%. In other words, smaller substrings are better recognized than large ones. This indicates that our proposed algorithm actually takes the structure of the state into account and learns to perform best rewarded actions according to this specific structure.

Q-net Activation Functions

As stated above, from all activation functions provided by the Tensorflow framework, we found the *tanh* function to yield the best results for our setting. Comparison of $R_2(x, a)$ to the baseline after a relatively short period of 1000 generations yielded the results as depicted in the following table, where the activation function $f(x)$ is given in the second row.



(a) (b)

Figure 8.5.: $E_2 - n_g - A$ diagram for $w = 32$, $\epsilon = 0.1$, and $\gamma = 0.2$ for $|A| = 8$. Figure (a) shows an overall initial decrease of the Q function depending on the number n_g of generations, followed by explorative behavior (b).

tanh	sigmoid	elu	softplus	softsign	relu
$\tanh(x)$	$(1+e^{-x})^{-1}$	$\begin{cases} \alpha(e^x - 1) & x < 0 \\ x & x \geq 0 \end{cases}$	$\ln(1+e^x)$	$\frac{x}{1+\lvert x\rvert}$	$\begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$
7.75%	6.56%	5.3%	2%	6.4%	1.3%

Learning Rates

As discussed in Section 8.4 the convergence rate of Q is primarily determined by the learning rate of stochastic gradient descent during back-propagation as well as the choice of γ. Increasing γ and gradient descent rate and comparing against the baseline shows that small values of γ result in less steep mean reward improvements compared to the baseline. A visualization for the Q function is depicted in Figure

State Generalization

In order to achieve high-throughput fuzzing we tested if the already trained Q net generalizes to previously unseen inputs. This would allow us to switch off Q net training after a while and therefore avoid the high processing costs of evaluating the coverage reward. To measure generalization we restricted the offset $o \in \{0, ..., |x| - |x'|\}$ in the training phase to values in the first half of the seed file. For testing, we omitted reward measurement in the Q update step as depicted in Figure 8.2 to stop the training phase and only considered offsets in the second half of the seed file. This way, the Q net is confronted with previously unseen states. This resulted in an improvement in execution time of 4.7% compared to the baseline. This shows that the algorithm generalizes to states in the sense that it still performs better than the baseline given previously unobserved states. This allows us to generate new mutations with a fixed Q function (corresponding to a fixed policy) so that we can still tolerate slow Q network training.

Effective Mutation Ratios

The defined action classes are quite different in nature and their effects on the state differ significantly. This rises the question if we can still achieve better results if we only allow very similar actions. In other words, is our reinforcement fuzzer able to distinct the effects of similar actions? Therefore, we restricted the actions A to random bit flips. We fixed 32 actions with 32 different mutation ratios $\{0.004 + k * 0.001 | k = 1, ..., 32\}$ to yield a 4.54% improvement compared to the baseline. This indicates that the Q network can still distinguish between very similar actions, although naturally not as good as in the setting of different action classes.

8.6. Discussion

The policy π as defined in Section 8.3 can be interpreted as a form of generalized grammar for the input structure. Given a specific state it provides a string replacement based on experience. Especially if we reward execution path depth, we indirectly reward validity of the input with regard to the defined input structure, as non-valid inputs are likely to be rejected early during parsing. From this perspective, the defined algorithm both generates input according to the rewriting rules and simultaneously adapting these rules based on successive feedback.

Further, current state-of-the art fuzzers define a fixed set of actions regardless of the input string to be mutated. Enhancing fixed-action fuzzers with our approach would require only small modifications to the mutator engines without changing the rest of the fuzzing framework. It remains an open question if Q-learning integration into existing frameworks will improve overall fuzzing quality or lower it due to performance overhead.

8.7. Conclusion

Inspired by the similar nature of feedback-driven random testing and reinforcement learning we introduce the first fuzzer that learns to perform highly rewarded mutations with respect to a predefined search heuristics. By automatically rewarding runtime characteristics of the target program we obtain inputs that likely drive program execution towards a predefined goal, e.g. maximization of code coverage or processing time. To achieve this we formulate fuzzing as a reinforcement learning problem based on the language of Markov decision processes as introduced in Part I of this work. This allows us to construct an algorithm based on deep Q-learning that learns to choose highly rewarded actions given an input seed. Adapting the Q function iteratively yields a policy that encodes the behavior of the Markov decision process. We indicate the feasibility of our approach by implementing a prototype targeted against PDF parsers.

9. Conclusion

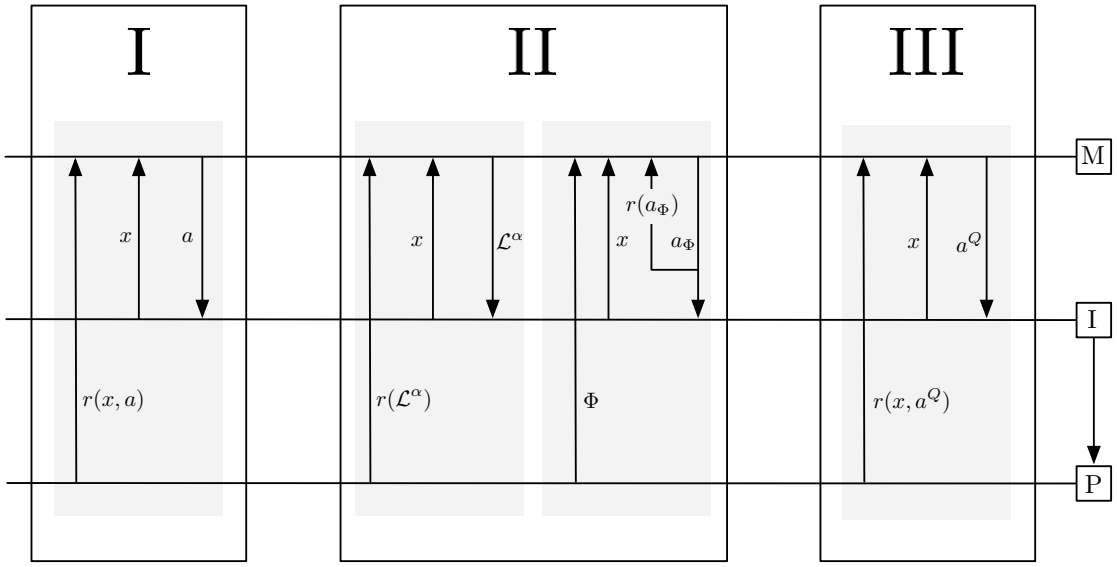


Figure 9.1.: Overview.

This chapter ends our journey through the world of advanced fuzzing. After a short recapitulation we return to the very beginning of this work and present the announced connection of our discoveries and embedding of our overall map into a global atlas. Putting our results in a slightly different context yields a dual view on the subject and gives an outlook for the challenges to come.

Motivated by the urgent need for automated software testing and the lack of probability theory in state-of-the-art random testing tools, we set the target to connect the deep knowledge of stochastic processes with software testing. We achieved this by modeling feedback-driven fuzzing in the language of Markov decision processes in Part I of this work. This formulation allowed us to translate processes with suitable characteristics into concrete fuzzing algorithms in Part II. By enhancing this stochastic approach with exact computation based on symbolic execution we were able to fuzz into deep layers of the targeted programs. In order to explore the full capabilities of our mathematical model, we made use of very recent results from the field of reinforcement learning. Similar to chess and the game of Go, we conjecture that machine learning techniques will continue to surpass purely human fuzzing strategies in the future.

An overview of this work is depicted in Figure 9.1. In Part I we introduce the language in which we formulate fuzzing with predefined behavior in Part II and fuzzing with

learning behavior in Part III. The mutator engine M generates an input I that is injected into the program under test P in order to observe and evaluate feedback from P during runtime. Each part of this work illuminates this feedback loop from a different perspective and with a variety of mathematical methods. As we went along we took up all fragments of Figure 9.1 and explained their meaning in detail.

9.1. Markov Decision Machines

Let us now look at fuzzing in another light. Similarly to the probabilistic Turing machine that operates on a set of symbols with predefined transition relations, our mathematical model of fuzzing gives rise to a Markov decision machine. In contrast to the probabilistic Turing machine, we do not define accepting final states and consider probabilistic translation relations that are dynamically adapted during tape processing instead of statically defined ones. In other words, our fuzzer directly implements a Semi-Thue system with dynamically adapted substitution rules. Instead of accepting states we have rewards that guide string substitution towards a predefined goal. The substitution rules are dynamic in the sense that their probability of execution as defined in Equation (4.4) is adapted by updating the policy π over time using the Q-learning approach (as defined in Section 8.3). The state of the Markov decision machine (not to be confused with the state x of the decision process) is indicated by its behavior learned over time, i.e. the sequence of kernels $(\pi_i)_{i\in\mathbb{N}}$ with $\pi_t = \pi_t(\cdot|x_0, a_0, r_0, ..., x_{t-1}, a_{t-1}, r_{t-1}, x_t)$. This dual view of our approach might provide new impulses in theoretical informatics.

9.2. Outlook

We embed our thoughts into a broader view on the subject and open the gates to two general directions for future research.

9.2.1. Hierarchies of Learning Agents

Let us place two illustrative examples first. In order to drive our fuzzers towards a predefined goal, we gave them rewards as discussed in Section 4.3 to award good mutation actions and sanction ineffective ones. With the help of machine learning techniques we were able to take performed actions as well as raw system states into account for learning rewarding policies and therefore establishing effective behavior. We applied deep neural networks to be able to deal with the complex state structures. However, the measurements for the concrete values were quite simple. We measured coverage of code, execution time, or the time it takes to perform the mutation. We did this because these values are correlated with finding bugs fast (as discussed in detail in Part I). How can we bridge those indirect measures and reward the presence or neighborhood of bugs directly? If we could find a way to detect characteristic conditions of certain bug classes during program execution, we could feedback this information as a reward. If we could further find some kind of similarity measure, we could define a distance to such bug conditions

and directly reward the fuzzer to generate inputs that drive the program towards the bug. Machine learning techniques helped us to deal with complex actions and system states in Part III. Intuitively, machine learning might also help us in detecting complex bug characteristics in the program flow.

Further, we could proceed to research the interaction between multiple fuzzers. In Chapter 7 we discussed an approach to organize large-scale fuzzing in computer clusters. In the spirit of Part III we could introduce learning techniques to this interaction. A swarm of fuzzers that is controlled by an overall learning agent might reveal efficient strategies for larger fuzzing campaigns.

Both examples, the sensing mechanism for characteristic bug classes and the controlling agent for a swarm of fuzzers, have one thing in common: They introduce machine learning into the overall testing process, the former via sub-agents within the feedback-loop, the latter via super-agents above multiple fuzzing instances. Considering such hierarchies of learning agents might be a fruitful new direction in software testing.

9.2.2. Alternative Models

The work at hand was motivated by two research challenges, namely to connect probability theory to software testing and to translate stochastic processes into fuzzing algorithms. We approached this adventure with constructing a mathematical model for fuzzing. All results of the work at hand are findings from explorations within this model. It is important to realize that this model is just one possibility within a manifold of alternatives. Other models will most likely reveal very different techniques. To illustrate this thought very impressively, let us have a look at the way we defined actions. In Section 4.2 we modeled actions as random variables mapping substrings of an input to probabilistic rewriting rules. Let us consider the following variant: What if actions could not only transform the input of the program under test, but the program itself? Imagine an action that transforms a program into representations that are more accessible to fuzzing. As an example, such an action could replace compare instructions with large operands by multiple comparisons with small operands, increasing the probability to pass them during input generation. Such actions that translate the target into equivalent representations open the door to completely undiscovered fields of research. Similarly, alternative mathematical models will reveal interesting new characteristics and novel approaches for testing.

This thesis was written following the principle of *"Calculemus!"* in the spirit of Gottfried Wilhelm Leibniz. We hope that the reader is inspired to further explore the power of mathematical modeling.

Bibliography

[1] Ian Hacking. *The emergence of probability: A philosophical study of early ideas about probability, induction and statistical inference.* Cambridge University Press, 2006.

[2] Alexander Kolmogoroff. *Grundbegriffe der Wahrscheinlichkeitsrechnung.* Springer, 1933.

[3] Jean Jacod and Philip Protter. *Probability essentials.* Springer Science & Business Media, 2004.

[4] Norbert Wiener. *Cybernetics: or Control and Communication in the Animal and the Machine*, volume 25. MIT press, 1961.

[5] Gerald Tesauro. Practical issues in temporal difference learning. In *Advances in neural information processing systems*, pages 259–266, 1992.

[6] Gerald Tesauro. Td-gammon: A self-teaching backgammon program. In *Applications of Neural Networks*, pages 267–285. Springer, 1995.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[9] Daniel Angermeier, Konstantin Böttinger, Andreas Ibing, Dieter Schuster, Frederic Stumpf, and Dirk Wacker. A secure architecture for smart meter systems. In Yang Xiang, Javier Lopez, C.-C.Jay Kuo, and Wanlei Zhou, editors, *Cyberspace Safety and Security*, volume 7672 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 2012.

[10] Frederic Stumpf and Konstantin Böttinger. When the lights go out - attacks and security solutions for smart metering. In *23. SmartCard Workshop 2013. Tagungsband : Darmstadt, 6./7. Februar 2013*, pages 158–169. Fraunhofer Verlag, 2013, 2013.

[11] Julian Horsch, Konstantin Böttinger, Michael Weiß, Sascha Wessel, and Frederic Stumpf. Trustid: Trustworthy identities for untrusted mobile devices. In *Proceedings of the 4th ACM conference on Data and Application Security and Privacy*, pages 281–288. ACM, 2014.

[12] Konstantin Böttinger, Dieter Schuster, and Claudia Eckert. Detecting fingerprinted data in TLS traffic. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 633–638, New York, NY, USA, 2015. ACM.

[13] Giuseppe Settanni, Florian Skopik, Helmut Kaufmann, Tobias Gebhardt, Klaus Theuerkauf, Konstantin Böttinger, Mark Carolan, Damien Conroy, and Pia Olli. A blueprint for a pan-european cyber incident analysis system. In *3rd International Symposium for ICS & SCADA Cyber Security Research 2015, ICS-CSR 2015*, 2015.

[14] Konstantin Böttinger, Gerhard Hansch, and Bartol Filipovic. Detecting and correlating supranational threats for critical infrastructures. In *15th European Conference on Cyber Warfare and Security (ECCWS 2016)*, 2016.

[15] Giuseppe Settanni, Florian Skopik, Yegor Shovgenya, Roman Fiedler, Mark Carolan, Damien Conroy, Konstantin Böttinger, Mark Gall, Gerd Brost, Christophe Ponchel, Mirko Haustein, Helmut Kaufmann, Klaus Theuerkauf, and Pia Olli. A collaborative cyber incident management system for European interconnected critical infrastructures. *Journal of Information Security and Applications*, Special Issue on ICS & SCADA Cyber Security, 2016.

[16] Konstantin Böttinger. Hunting bugs with Lévy flight foraging. In *IEEE Symposium on Security and Privacy Workshops 2016*, pages 111–117, 2016.

[17] Konstantin Böttinger and Claudia Eckert. Deepfuzz: Triggering vulnerabilities deeply hidden in binaries. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 25–34. Springer, 2016.

[18] Konstantin Böttinger. Fuzzing binaries with Lévy flight swarms. *EURASIP Journal on Information Security*, 2016.

[19] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? The impact of copy&paste on android application security. In *IEEE Symposium on Security and Privacy 2017*, 2017.

[20] Konstantin Böttinger. Guiding a colony of black-box fuzzers with chemotaxis. In *IEEE Symposium on Security and Privacy Workshops 2017*, 2017.

[21] Konstantin Böttinger. Chemotactic test case recombination for large-scale fuzzing. *Journal of Cyber Security and Mobility*, pages 269–286, 2017.

[22] Konstantin Böttinger. Hunting bugs with nature-inspired fuzzing. In *Nature-inspired Cyber Security and Resilience: Fundamentals, Technology and Applications*. The Institution of Engineering and Technology, 2018.

[23] Konstantin Böttinger, Rishabh Singh, and Patrice Godefroid. Deep reinforcement fuzzing. In *IEEE Symposium on Security and Privacy Workshops 2018*, 2018.

[24] Peter Schneider and Konstantin Böttinger. High-performance unsupervised anomaly detection for cyber-physical system networks. In *ACM Workshop on Cyber-Physical Systems Security Privacy (CPS-SPC)*, 2018.

[25] Peggy Aldritch Kidwell and Michael R Williams. The calculating machines: their history and development. *Massachusetts Institute of Technology and Tomash Publishers, USA*, 1992.

[26] Alan Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, England, 1949.

[27] F Lockwood Morris and Clifford B Jones. An early program proof by alan turing. *IEEE Annals of the History of Computing*, 6(2):139–143, 1984.

[28] Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, 1966.

[29] Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[30] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[31] Joseph Ruthruff, Robert C Armstrong, Benjamin Garry Davis, Jackson R Mayo, and Ratish J Punnoose. Leveraging formal methods and fuzzing to verify security and reliability properties of large-scale high-consequence systems. Technical report, Sandia National Laboratories (SNL-CA), Livermore, CA (United States), 2012.

[32] Gerald M Weinberg. *Perfect software: And other Illusions about testing*. Dorset House Publishing Co., Inc., 2008.

[33] Peter H Salus and G Vinton. *Casting the Net: From ARPANET to Internet and Beyond...* Addison-Wesley Longman Publishing Co., Inc., 1995.

[34] Katie Hafner and Matthew Lyon. *Where wizards stay up late: The origins of the Internet*. Simon and Schuster, 1998.

[35] Hilarie Orman. The morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, 99(5):35–43, 2003.

[36] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, July 1984.

[37] Joe W Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183. IEEE Press, 1981.

[38] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[39] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[40] Klaus Wüst. *Mikroprozessortechnik: Grundlagen, Architekturen, Schaltungstechnik und Betrieb von Mikroprozessoren und Mikrocontrollern*. Springer-Verlag, 2010.

[41] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*. Intel Corporation, 2016.

[42] P Daniel, Cesati Marco, et al. Understanding the linux kernel, 2007.

[43] Donald Lewine. *POSIX programmers guide*. " O'Reilly Media, Inc.", 1991.

[44] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[45] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: the past, the present, and the future. *Research in Attacks, Intrusions, and Defenses*, pages 86–106, 2012.

[46] C Vanden Berghe, James Riordan, Frank Piessens, et al. A vulnerability taxonomy methodology applied to web services. In *Proceedings of the 10th Nordic Workshop on Secure IT Systems (NordSec 2005)*, pages 49–62, 2005.

[47] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6), 2006.

[48] László Szekeres, Mathias Payer, Tao Wei, and Dong Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[49] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[50] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[51] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, Boston, MA, USA, 1 edition, 2007.

[52] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[53] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[54] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[55] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000.

[56] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[57] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[58] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[59] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[60] Dirk Werner. *Funktionalanalysis*. Springer, 2006.

[61] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 391–402, New York, NY, USA, 2008. ACM.

[62] James Clause and Alessandro Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 249–260, New York, NY, USA, 2009. ACM.

[63] Matthias Höschele and Andreas Zeller. Mining input grammars with autogram. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 31–34, Piscataway, NJ, USA, 2017. IEEE Press.

[64] David Leon and Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 442–456, Washington, DC, USA, 2003. IEEE Computer Society.

[65] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–188, Washington, DC, USA, 1999. IEEE Computer Society.

[66] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 861–875, Berkeley, CA, USA, 2014. USENIX Association.

[67] Friedrich Lenz, Thomas C Ings, Lars Chittka, Aleksei V Chechkin, and Rainer Klages. Spatiotemporal dynamics of bumblebees foraging under predation risk. *Physical review letters*, 108(9):098103, 2012.

[68] Gandhimohan M Viswanathan. Ecology: Fish in Lévy-flight foraging. *Nature*, 465(7301):1018–1019, 2010.

[69] Nicolas E Humphries, Nuno Queiroz, Jennifer RM Dyer, Nicolas G Pade, Michael K Musyl, Kurt M Schaefer, Daniel W Fuller, Juerg M Brunnschweiler, Thomas K Doyle, Jonathan DR Houghton, et al. Environmental context explains Lévy and Brownian movement patterns of marine predators. *Nature*, 465(7301):1066–1069, 2010.

[70] Deborah Austin, W. D. Bowen, and J. I. McMillan. Intraspecific variation in movement patterns: modeling individual behaviour in a large marine predator. *Oikos*, 105:15–30, 2004.

[71] Gabriel Ramos-Fernández, José L Mateos, Octavio Miramontes, Germinal Cocho, Hernán Larralde, and Barbara Ayala-Orozco. Lévy walk patterns in the foraging movements of spider monkeys (Ateles geoffroyi). *Behavioral Ecology and Sociobiology*, 55(3):223–230, 2004.

[72] Gandhimohan M Viswanathan, V Afanasyev, SV Buldyrev, EJ Murphy, PA Prince, H Eugene Stanley, et al. Lévy flight search patterns of wandering albatrosses. *Nature*, 381(6581):413–415, 1996.

[73] Anders Mårell, John P. Ball, and Annika Hofgaard. Foraging and movement paths of female reindeer: insights from fractal analysis, correlated random walks, and Lévy flights. *Canadian Journal of Zoology-revue Canadienne De Zoologie*, 80:854–865, 2002.

[74] O Bénichou, C Loverdo, M Moreau, and R Voituriez. Intermittent search strategies. *Reviews of Modern Physics*, 83(1):81, 2011.

[75] Tajie H Harris, Edward J Banigan, David A Christian, Christoph Konradt, Elia D Tait Wojno, Kazumi Norose, Emma H Wilson, Beena John, Wolfgang Weninger, Andrew D Luster, et al. Generalized Lévy walks and the role of chemokines in migration of effector CD8+ T cells. *Nature*, 486(7404):545–548, 2012.

[76] Gandhimohan M Viswanathan, Marcos GE Da Luz, Ernesto P Raposo, and H Eugene Stanley. *The physics of foraging: an introduction to random searches and biological encounters*. Cambridge University Press, Cambridge, England, 2011.

[77] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, Inc., New York, NY, USA, 1999.

[78] Sang Kil Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 725–741, May 2015.

[79] V. Zaburdaev, S. Denisov, and J. Klafter. Lévy walks. *Rev. Mod. Phys.*, 87:483–530, Jun 2015.

[80] Marie Chupeau, Olivier Bénichou, and Raphaël Voituriez. Cover times of random searches. *Nature Physics*, 11:844–847, 2015.

[81] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[82] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[83] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2:215–222, 1976.

[84] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[85] Patrice Godefroid. Compositional dynamic test generation. In *ACM SIGPLAN Notices*, volume 42, pages 47–54. ACM, 2007.

[86] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69, September 2011.

[87] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40, March 2012.

[88] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[89] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[90] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[91] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11(4):339–353, 2009.

[92] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering, 2007. ICSE 2007.*, pages 416–426. IEEE, 2007.

[93] Antonio Filieri, Corina S. Păsăreanu, Willem Visser, and Jaco Geldenhuys. Statistical symbolic execution with informed sampling. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 437–448. ACM, 2014.

[94] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176. ACM, 2012.

[95] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S. Păsăreanu. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 866–877. ACM, 2015.

[96] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[97] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for automated adversarial testing of certificate

validation in SSL/TLS implementations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 114–129, Washington, DC, USA, 2014. IEEE Computer Society.

[98] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 793–804, New York, NY, USA, 2015. ACM.

[99] Jesús A De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of symbolic computation*, 38(4):1273–1302, 2004.

[100] Matthias Köppe. A primal Barvinok algorithm based on irrational decompositions. *SIAM Journal on Discrete Mathematics*, 21(1):220–236, 2007.

[101] JA De Loera, Brandon Dutra, Matthias Koeppe, Stanislav Moreinis, Gregory Pinto, and Jianqiu Wu. Software for exact integration of polynomials over polyhedra. *ACM Communications in Computer Algebra*, 45(3/4):169–172, 2012.

[102] Jonathan Salwan and Florent Saudel. Triton: A concolic execution framework for x86-64 binaries. In *Symposium sur la securite des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[103] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[104] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[105] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *26th 2011 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 53–62, Nov 2011.

[106] K. Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 297–306, Washington, DC, USA, 2008. IEEE Computer Society.

[107] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *IEEE/IFIP International Conference on Dependable Systems and Networks DSN'09*, pages 359–368. IEEE, 2009.

[108] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, February 2014.

[109] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *NDSS*, volume 11, pages 59–66, 2011.

[110] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 511–522, New York, NY, USA, 2013. ACM.

[111] Joe R Riley, Uwe Greggers, Alan D Smith, Don R Reynolds, and Randolf Menzel. The flight paths of honeybees recruited by the waggle dance. *Nature*, 435(7039):205–207, 2005.

[112] David JT Sumpter and Madeleine Beekman. From nonlinearity to optimality: pheromone trail foraging by ants. *Animal behaviour*, 66(2):273–280, 2003.

[113] Kevin J Painter and Thomas Hillen. Volume-filling and quorum-sensing in models for chemosensitive movement. *Can. Appl. Math. Quart*, 10(4):501–543, 2002.

[114] Thomas Hillen and Kevin J Painter. A user's guide to pde models for chemotaxis. *Journal of mathematical biology*, 58(1-2):183–217, 2009.

[115] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, pages 138–157, 2016.

[116] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[117] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[118] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[119] Patrice Godefroid, Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan. Test Generation Using Symbolic Execution. In *FSTTCS*, volume 18, pages 24–33. Citeseer, 2012.

[120] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 95–110, New York, NY, USA, 2017. ACM.

[121] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[122] CJCH Wattkins. *Learning from delayed rewards*. PhD thesis, Cambridge University, 1989.

[123] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, January 2017.

[124] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.