Experiences in using Reinforcement Learning for Directed Fuzzing.

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Technology (Research)

IN THE

Faculty of Engineering

BY

Subhendu Malakar

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

December, 2019

Declaration of Originality

I, **Subhendu Malakar**, with SR No. **04-04-00-10-22-16-1-13991** hereby declare that the material presented in the thesis titled

Experiences in using Reinforcement Learning for Directed Fuzzing

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2016-2019**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Sunday 29th December, 2019 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. Vinod Ganapathy Advisor Signature

DEDICATED TO

My parents,
and my niece, Krittika.

Acknowledgements

First of all, I would like to thank my parents for supporting me in my decisions. Leaving a job and coming back for higher studies is always a hard decision to make. You guys made it simple, and this would not have been possible without your support and care. Next, I would like to thank my advisor Dr Vinod Ganapathy, to guide me through this journey. His continuous support and belief in me helped me immensely in times when I was not sure about my work. He was very patient with my work and always ready to discuss solutions. Will definitely miss those brainstroming sessions and “not so daily” scrums.

Next, I would like to thank my friends Sagar, Monika, Soham, and Aritra. Sagar, for always listening to my complaints, and be there in both happy and sad times. Monika, thank you very much for the technical discussions and advice. Hope I didn’t exhaust both of you guys with my tantrums. Soham for his conducive inputs in my research and Aritra for his constant sarcastic encouragement. I would also like to thank my labmates Abhishek, Aditya, Ajay, Arun, Kripa, Rakesh, Rounak, and Sridhar for making the lab a fun place.

This list will be incomplete without mentioning the non-teaching staff of CSA. Everyday they provide a friendly and clean environment for us and make sure that we are not distracted. Special mention of Sudesh bhaiya, an always smiling face in the morning goes a long way.

Lastly, I would like to thank my niece, Krittika, for keeping the kid in me alive. You are too young (3.5 years old at the time of writing) to understand any of this, but listening to your stuttering words and report of your mischevious acts always put a smile.

Abstract

Directed testing is a technique to analyze user-specified target locations in the program. It reduces the time and effort of developers by excluding irrelevant parts of the program from testing and focusing on reaching the target location. Existing tools for directed testing employ either symbolic execution with heavy-weight program analysis or fuzz testing mixed with hand-tuned heuristics.

In this thesis, we explore the feasibility of using a data-driven approach for directed testing. We aim to leverage the data generated by fuzz testing tools. We train an agent on the data collected from the fuzzers to learn a better mutation strategy based on the program input. The agent then directs the fuzzer towards the target location by instructing the optimal action for each program input. We use reinforcement learning based algorithms to train the agent. We implemented a prototype of our approach and tested it on synthetic as well as real-world programs. We evaluated and compared different reward functions.

In our experiments, we observe that for simple synthetic programs, our approach can reach the target location with fewer mutations compared to AFL and AFLGo that employ random mutations. However, for complex programs, the results are mixed. No one technique can perform consistently for all programs. For real-world programs, our approach failed to find an input that reaches the target location.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Software testing is the process of finding bugs and vulnerabilities in a given software application. The process usually involves generating inputs for the program such that the generated inputs cause a crash due to a bug in the program. There are a variety of tools developed for software testing. All these tools generally focus on "whole-program" testing, i.e., they do not differentiate between different portions of the program. However, often, we need to focus only on a specific portion of the program. For example, these "interesting" portions can be a parser which validates the input from the user before passing it for further computation, or it can be newly added code for a new feature or a patch to fix some bug. This kind of testing is referred to as directed testing. In directed testing, only specific "interesting" portions of the program are analyzed for buggy behaviour. These "interesting" portions are either specified by the user or automatically deduced by other static analysis tools. Since programs are written incrementally, focusing on testing the altered or newly added sections in the program would be judicious use of time and resources.

Symbolic execution is the most popular technique for directed testing [1, 2]. It is a way of analyzing programs by inferring a logical formula representing program execution, based on concrete execution. Symbolic execution involves using *heavy-weight* program analysis and SMT solvers. To generate an input that reaches the target location, tools based on symbolic execution first need to collect all the path constraints leading to the target location. All of these path constraints are satisfied by all the possible inputs that reach the target location following that path. The path constraints thus collected are passed to a constraint solver like Z3 [3] that attempts to generate a concrete program input satisfying all the path conditions. The generated concrete program input is guaranteed to reach the target location.

Ma et al. [1] cast directed testing as a line reachability problem, i.e., given a target statement in the program, the goal is to find a realizable path to the target. They have

proposed multiple directed symbolic execution search strategies for reachability. These search strategies differ in the order of collection of the path constraints. KATCH [2] is the state of the art tool for patch testing based on symbolic execution. It starts by selecting an input from the existing test cases of the program's regression suite based on its *estimated distance* to the patch. It then combines symbolic execution with three heuristics based on program analysis to derive a new input that reaches the target location.

Although symbolic execution, in theory, is capable of finding inputs that reach the target location, it has several limitations. Any symbolic execution driven tool has to iterate over all the feasible paths in the program. The number of such paths grows exponentially as the program grows and can be infinite in some instances, e.g., when the program has unbounded loops. Even in the case of directed testing, where the search space for the paths reaching the target location is significantly reduced, the problem persists. The program analysis employed is *heavy-weight* as in that requires a detailed understanding of the underlying language semantics and memory models. It also requires accurate translation of each path condition to its corresponding constraint. The SMT solvers employed are dependent on the underlying theories and theories for many program constructs are not always available. Moreover, even when the theories are available, SMT solvers are known to take a considerable amount of time to come up with a solution. For example, real-world programs often involve non-linear arithmetics, which are undecidable for the solver. These limitations hamper the use of tools based on symbolic execution for large programs. For example, KATCH [2] was able to increase the patch coverage considerably, yet the authors claim that the tool was not able to find most of the targets and that more advances are needed to realize the goal of fully automated patch testing.

Recently, fuzz testing has gained much traction in the community for its simplicity, scalability and effectiveness to find bugs [4, 5, 6, 7, 8, 9]. Fuzz testing (or fuzzing) is an automated process of testing a program by providing it with random inputs. The user provides an initial set of inputs, called seeds, to the fuzzing tool (called fuzzer). Fuzzers then generate more inputs from these seed inputs by mutating them. These mutations range from single-bit operations like bit flip to single-byte mutations like byte flip, addition/subtraction to multi-byte mutations like insert bytes to or delete bytes from a location. Fuzzers typically do not employ heavy analysis on the program which makes them lightweight and thus scalable. Popular fuzzers like AFL [4] and libfuzzer [5] have found numerous bugs in large programs [10, 11].

Fuzzers are generally used to find bugs in the whole program. The main objective of any such generic fuzzer is code coverage, i.e., to cover as much code as possible. Since these generic fuzzers do not differentiate between paths, they are not incentivized to reach

a particular location in the program. It makes the generic fuzzers unsuitable for directed testing as they would unnecessarily mutate irrelevant inputs. Recent works [12, 13] have tried to modify these generic fuzzers for directed testing.

AFLGo [12] is the first such attempt to adapt generic fuzzers for directed testing. It is an extension of AFL [4] for directed greybox fuzzing. In greybox fuzzing, the fuzzers apply lightweight techniques, like program instrumentation, to approximate the internal structure of the program. AFLGo casts reachability as an optimization problem and tries to minimize the *seed distance*. Seed distance for a program input is defined as the average of the distances between each executed basic block in the program and the target basic block. The distance computed for each basic block is based on the intra-procedural control-flow graph (CFG) of each function, and the call graph (CG) of the whole program. Each basic block's distance to the target location is computed and instrumented during compile time. The seed distance, thus calculated, determines the energy of the seed. The energy of a seed is a measure of time the fuzzer spends mutating it. AFLGo employs *simulated annealing* as a *meta-heuristic* to minimize the seed distance. *Simulated annealing* is implemented as a power schedule, which controls the energy of all seeds.

Hawkeye [13] improves upon both static and dynamic analyses done by AFLGo. It presents a robust distance-based mechanism for the directed fuzzer by taking all the paths to the target location into account. During runtime, Hawkeye categorizes seeds into three queues based on the seed distance and covered function similarity. Based on the queue, Hawkeye prioritizes and schedules each seed for mutation. It also adopts an adaptive mutation strategy based on the seed distance. It prioritizes fine-grained mutation for inputs which are closer to the target location and vice-versa.

All the approaches mentioned earlier are heavily engineered and tweaked for directed testing. KATCH employed multiple heuristics based on program analysis to complement symbolic execution and mitigate some of its shortcomings. AFLGo had to come up with a specific meta-heuristic like simulated annealing to minimize the seed distance. Hawkeye also adopts various fine-tuned strategies for best results. All of these approaches try to find the best possible combination of tweaks needed for directed testing. These tweaks work for most of the programs but need scrupulous engineering.

To mitigate the task of trying to find the best heuristic, this thesis seeks to explore a data-driven approach for directed testing. Recent advances in fuzzing have used machine learning techniques to learn the grammar from the inputs [14], increase the efficiency of fuzzers [15], and learn about the optimal locations in the input files to mutate [16]. In this thesis, we explore the application of reinforcement learning for directed testing. There

are several reasons that intuitively suggest that reinforcement learning is a good fit for this domain, and we intend to evaluate whether these intuitions hold empirically. Reinforcement learning (RL) is an area of machine learning which deals with an agent and its interaction with the environment. RL agents have been used extensively in game environments where they have beaten world champions in the game of Go, Chess and Atari [17, 18, 19]. RL agents have also been used in robot control and traffic light management [20, 21].

An RL agent is an entity which learns about its environment by taking actions and observing the result. An RL environment is modelled as Markov Decision Process (MDP) which formulates decision making in a stochastic process. In an RL environment, the agent's goal is to maximize the cumulative reward until it reaches the final state. The final state is a unique state in the environment, where after reaching, the agent doesn't need to take any further actions. A robot reaching the target location on the floor, or a player winning in the game of chess or pong are some example of the final state in their respective environment. The agent chooses an action for each state to reach the next state and receives a corresponding reward for that action. For example, in a game of chess, the agent has to choose a piece to make a move. In a robot manoeuvering environment, the agent has to choose to either go right or left or go keep moving forward to reach the target location. A reward is a scalar quantity which approximates the $goodness$ of the action for that state. A positive reward means that the action taken resulted in the agent getting closer to its goal state. Similarly, a negative reward may mean that the action taken by the agent leads it away from the goal state. Winning a game of chess can be treated as a positive reward and losing the game incurs a negative reward. Similarly, a robot reaching its destination gets a positive reward and crashing into the wall receives a negative reward. Based on the experiences the agent gains from the exploration, the agent learns to choose the optimal action for each state to maximize the cumulative reward.

In this thesis, we attempt to adopt reinforcement learning algorithms for directed fuzzing. In our context, we model the fuzzer as an RL agent and the program under test as the environment. Each program input is an agent state, and the set of mutations on the program input are the possible actions for the agent on that state. So, the search space for the agent is the set of all possible inputs to the program. The final state or the goal state is any program input which reaches the target location in the program.

Thus, the problem can now be pictured as an agent trying to find a program input which reaches the target location by continuously mutating the present program input. The agent moves from one state to another and accumulates the reward in each transition. Since the maximum reward earned by an agent is bounded, the agent learns to avoid negative rewards,

i.e., mutations that deviate it from the target location. Rewards are the only source of information for the agent to learn and thus, it is essential to choose a useful reward function for the agent. We propose three different kinds of rewards (§3.4.3) for the agent and compare their performance (§4.3). To train our agent, we have used three state-of-the-art RL algorithms, namely Deep Q-Network (DQN) [19], Double Duelling deep Q-Network (DDQN) [22], and Asynchronous Advantage Actor-Critic (A3C) [23] network.

In our experiments (§4.2), an agent trained with DDQN algorithm outperformed other agents based on DQN and A3C. On synthetic benchmarks, our approach was able to reach the target location in all instances. In some programs, our approach was able to reach the target location with fewer mutations than the state of the art fuzzers. However, in terms of the time taken to find an input that reaches the target location, our approach was significantly slower. This overhead is expected as the fuzzer waits for the agent's response before every mutation.

1.1 Contribution of the Thesis

The main contribution of this thesis are summarized as follows:

1. We explore a data-driven approach for directed fuzz testing.
2. We propose an approach using reinforcement learning to train an agent to learn a better mutation strategy.
3. We implemented and evaluated our proposed approach against the state of the art directed fuzzers.
4. We trained the agent with various state-of-the-art RL algorithms and measured their effectiveness for program input mutation.
5. We have evaluated and compared various reward functions for the agent.

1.2 Outline of the Thesis

This thesis is organized as follows:

- **Chapter 2:** It provides an overview of the techniques used in the thesis. We start by explaining the working of American Fuzzy Lop, a popular greybox fuzzer. Next, we explain different components of reinforcement learning and how it works.

- **Chapter 3:** Here, we describe the design of our approach. We discuss all the components of the tool and its functionality.
- **Chapter 4:** In this chapter we evaluate our approach and detail our findings.
- **Chapter 5:** We describe related work in this area. We discuss each technique, its limitations and how it differs from our approach.
- **Chapter 6:** In this chapter we conclude our work and provide directions for future work.

Chapter 2

Background

2.1 Fuzzing

Fuzz testing (or fuzzing) is an automated process of software testing by providing the program with random inputs. Fuzzing is one of the best tools available for detecting vulnerabilities in a program. The underlying concept of fuzzing is to generate inputs for the program that can trigger a bug in the program. Since its introduction, a substantial amount of effort has been made to increase the efficiency of these fuzz testing tools (called fuzzer). Fuzzers differ in the choice of technique that is used to generate inputs to the program.

Based on the awareness of the program under test, fuzzers are classified as black-box, grey-box, and white-box fuzzers. Black-box fuzzers like zzuf [24] have no information about the program under test. It generates new inputs for the program in a purely random manner and observes the results. Black-box fuzzers are very fast in generating new program inputs, but lack in producing useful program inputs. On the other hand, white-box fuzzers like DART [25] collect knowledge about the inner working of the program. To gather information, it analyses the binary or the source code of the program. Often it also inspects the runtime behavior of the program to generate efficient program inputs. White-box fuzzers produce useful program inputs but are very slow in generating new program inputs.

Grey-box fuzzers try to create a balance between speed and effectiveness. It collects little information about the program under test and utilizes it to generate useful inputs to the program. Grey-box fuzzers often instrument the program to collect runtime information like line coverage. Grey-box fuzzers are very effective in finding bugs in the program and are often preferred over black-box and white-box fuzzers. In the next section, we describe the working of one such grey-box fuzzer, American Fuzzy Lop (AFL) [4].

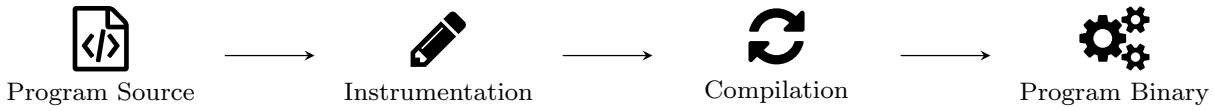


Figure 2.1: Block diagram of compilation phase of AFL

2.2 American Fuzzy Lop (AFL)

AFL [4] is the state-of-the-art grey-box fuzzer. The primary objective of AFL is to generate program inputs such that it increases the code coverage of the program under test. AFL employs genetic mutation strategies to generate new inputs. It employs minimal compile-time instrumentation to gather runtime information for each program input. Based on the runtime information, it decides whether to preserve the input for further mutations. AFL is easy to use, scalable, and its effectiveness in finding numerous bugs [10] in a variety of programs has gained it much popularity.

AFL works in two phases, a compilation phase, and a fuzzing phase. The compilation phase is responsible for adding instrumentation to the program source and building an instrumented program binary. The fuzzing phase is responsible for the generation of new program inputs and look for interesting behavior. We describe each phase in detail in the following subsections.

2.2.1 Compilation Phase

Every input to the program exercises an execution path in the program. An execution path is defined as a feasible sequence of basic blocks of the program. AFL instruments each basic block in the program to collect information about the execution path. This runtime information is used during the fuzzing phase to prioritize certain inputs.

To collect this runtime information, AFL identifies each basic block in the program with a random integer determined during compilation. Each branch is identified as a tuple (branch source, branch target). Since AFL captures branch coverage, it only needs to store the branch tuples. AFL computes the branch identifier as the output of the exclusive disjunction operator (XOR) of the branch tuple. To maintain these branch identifiers, AFL creates an array as shared memory where each byte corresponds to a branch identifier. A non-zero byte in the array signifies that the branch was hit during execution and the value of the byte represents the count of the number of times the branch got executed. The shared memory is attached to the instrumented binary during runtime. The fuzzer resets the shared memory before every execution. Note that AFL only stores the set of basic block executed and the sequence of the

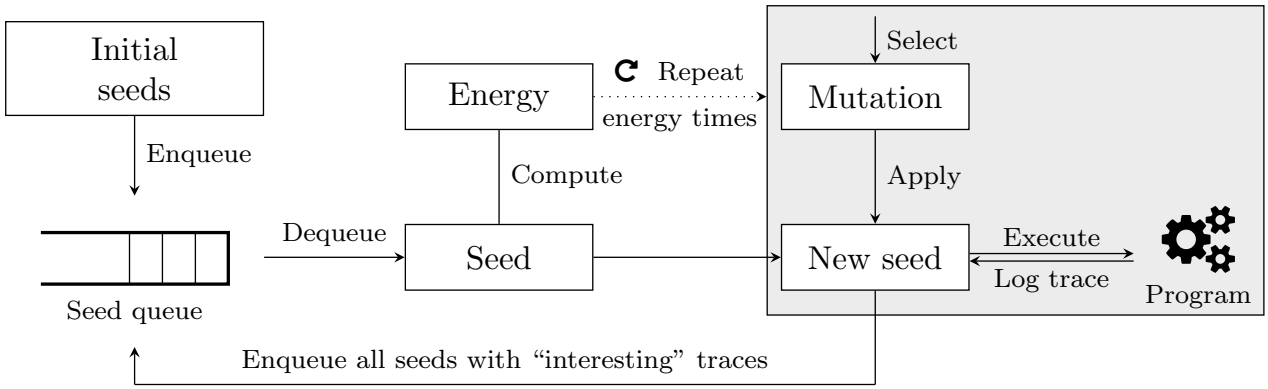


Figure 2.2: Block diagram of fuzzing phase of AFL

basic blocks executed is neglected.

The instrumentation added to each basic block is equivalent to:

$$\begin{aligned}&\text{cur\_location} = < \text{Compile Time Random Integer} >;\\ &\text{shared\_mem[cur\_location} \hat{} \text{ prev\_location]++;}\\ &\text{prev\_location} = \text{cur\_location} \gg 1;\end{aligned}$$

Here, $cur\_location$ denotes the identifier associated with each basic block. $prev\_location$ stores the identifier of the last basic block executed by the program. Before executing a basic block, the array in the shared memory region ($shared\_mem$) is updated based on the $cur\_location$ and $prev\_location$. $prev\_location$ is updated to $cur\_location$ bit shifted by 1. The shift operator serves two purposes. First, it preserves the directionality of the branch as the XOR operator is commutative. Second, it retains the identity of small loops. It is relevant for loops with a single basic block as the XOR operator nullifies the identifier (as x^x = 0), and multiple loops with a single basic block can be distinguished.

AFL uses LLVM compiler architecture to instrument the program. LLVM provides easy access to each basic block and APIs to add instrumentation. After instrumenting each basic block, the program is compiled normally, and the instrumented program binary is generated. Figure 2.1 shows the flow in the compilation phase of AFL. The instrumented program binary is used to get the runtime information during the fuzzing phase.

2.2.2 Fuzzing Phase

Fuzzing phase is the core of AFL, and it is responsible for the generation of new inputs. Algorithm 1 describes the procedure and figure 2.2 shows the flow in this phase. Every fuzzer need some program inputs, called seed inputs or seed, from the user to start fuzzing. AFL

initializes the $Queue$ with these user-provided seed inputs. The $Queue$ maintains the interesting seed inputs during fuzzing session. The fuzzer selects a seed from the $Queue$ for further mutations. It calculates the energy of each seed which defines the number of times the seed is mutated. It is calculated based on execution time, branch coverage, and freshness. The freshness of a seed is a representation of the number of times the seed has been fuzzed already. An input that is not yet fuzzed is fresh and preferred over another input which has been fuzzed multiple times. Also, seeds with less execution time and more branch coverage get higher energy.

Algorithm 1: American Fuzzy Lop - Fuzzing Phase Algorithm

Input: Program $\mathcal{P}$, Program arguments $\mathcal{A}$, Seed Inputs $\mathcal{S}$
Output: Interesting Seeds $\mathcal{Q}$

```
1  Function AFL_Fuzz_Loop(P, A, S)
2      Q ← φ
3      foreach seed ∈ S do
4          add_to_queue(Q, seed)
5      end foreach
6      while True do
7          seed ← select_seed(Q)
8          energy ← calculate_score(seed)
9          for i ← 1 to energy do
10             mutation ← select_mutation()
11             seed' ← mutate_seed(seed, mutation)
12             status ← run_target(P, A, seed')
13             if status == INTERESTING then
14                 add_to_queue(Q, seed')
15             end if
16         end for
17     end while
18     return Q
19 end Function
```

After selecting a $seed$ and calculating its energy, the fuzzer starts to mutate the $seed$. The mutation operations range from bit level mutations to byte level mutations, and multi-byte level mutations also. Bit level mutations include flipping a bit, a nibble, or consecutive bytes. Byte level mutations include addition or subtraction of random bytes at random locations.

It also involves overwriting bytes with some *interesting* bytes and generally represents the edge cases. Examples of such *interesting* bytes are 0, 1, 100, 32678, and -1. AFL also has insertion and deletion operations that insert or delete multiple bytes from random location in the program input. AFL applies a sequence of these mutation operations (termed stacking) on each seed to generate new program inputs. The instrumented program binary is executed with each new input $seed'$ and checked for interesting behaviors. If the newly generated seed is interesting, it is saved in the $Queue$ for further mutations.

An input is said to be interesting if it either executes a new branch or changes the execution count of some branch substantially. To check for interesting behavior, AFL maintains a global array, $virgin\_bits$. $virgin\_bits$ aggregates the hit count of each branch of the program by all the seed inputs. After each execution of the instrumented program binary with a new seed, the shared memory is compared with $virgin\_bits$ for interesting behavior.

Interesting program inputs execute previously unexplored branches and thus mutating them is more likely to generate more interesting program inputs. AFL maintains and mutates only the interesting seed inputs to maximize its chances of increasing the branch coverage. AFL also maintains the efficacy of each seed in the $Queue$ and periodically removes ineffective seeds from the $Queue$. One more fuzzing strategy that is effective is the splicing operator. When AFL gets stuck in finding new interesting seeds, it splices two seeds at random locations to generate a new program input and applies stacked mutations on it. The low-level compile-time instrumentation and low runtime overhead mixed with such heuristics and optimizations make it very fast and effective to find bugs in programs.

2.3 Reinforcement Learning : Basics

Reinforcement learning (RL) is a field in machine learning that involves learning from interactions. The learner (or agent) has to learn the optimal action to perform in any situation such that it maximizes the cumulative reward. The agent has no information about the optimal actions, and it learns this mapping by continuous interaction with the environment. RL agents are known to learn complex games like chess, Go, and Atari and have defeated human world champions [17, 18, 19]. RL agents are not just confined to game environments and are used in robot control and traffic light management also [20, 21]. We describe the agent's learning process in section 2.3.2.

Reinforcement learning is different from supervised learning, which is the most prevalent form of machine learning. In supervised learning, the learner is given a set of labeled examples to train itself. These labeled examples are in the form of feature values (state) and the corresponding labels (actions). The labels provide precise information about the target value

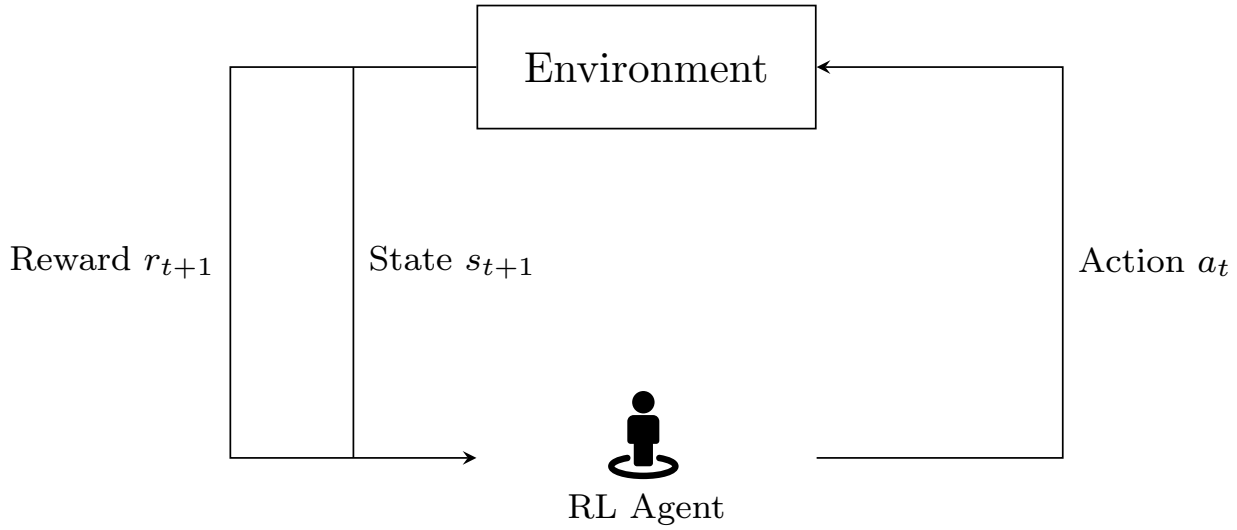


Figure 2.3: Reinforcement learning - Block Diagram

to be learned by the learner. Classification tasks like classifying an image as a picture of a cat or a dog fall under supervised learning. On the other hand, an RL agent doesn't know about the optimal action for a given situation. There is no $Oracle$ to provide the RL agent with this information. To learn this information, the RL agent explores the environment and learns from its own experiences.

Reinforcement learning is also different from unsupervised learning, which is mainly used to find hidden patterns in the data. In unsupervised learning, the learner is given a set of unlabeled data to find patterns, associations, or detect anomalies. Clustering algorithms, like aggregating customers based on purchasing behaviour, is an example of unsupervised learning. It seems similar to reinforcement learning, as in both cases, the learner is not provided with prelabeled output data to train itself. However, in reinforcement learning, the learner uses the output (reward) as a metric to evaluate its choice of action. This self-feedback mechanism is not available in unsupervised learning.

2.3.1 Reinforcement Learning : Components and Elements

There are two components in reinforcement learning, the agent, and the environment.

- **Agent**: An agent in reinforcement learning context is a learner who is tasked with the purpose of understanding the environment. An agent decides which action to choose in a situation. In a game of chess, a computer player is an agent which plays the game by choosing to play a chess piece. Initially, the agent doesn't know the rules of chess and how the game progresses.

- **Environment**: In reinforcement learning, an environment is a system which the agent seeks to understand. It consists of everything apart from the agent. The functioning of the environment is not known to the agent a priori. For example, in a game of chess, the chessboard and the opponent are part of the environment.

Figure 2.3 describes the interaction between the agent and the environment. Apart from the components, other elements in reinforcement learning are discussed below.

- **State**: A state is a representation of the environment's present situation. The position of each chess piece on the chessboard is the state of the environment.

- **Reward Signal**: After each interaction, the agent receives a numerical value from the environment. This numerical value, called reward signal or reward, describes the result of the action as good or bad. A high positive reward means that the action chosen by the agent was right, and a negative reward means that the action was terrible. The goal of an agent is to maximize the cumulative reward it receives from the environment over a period of time. The reward signal can be modeled differently for the same environment. For example, in a game of chess, an RL agent can be rewarded each time it takes a piece of the opponent and penalized if the opponent takes the agent's piece. Here we reward the agent for each take. Alternatively, the agent can be rewarded only when it is able to win the game and penalized for losing the game.

- **Policy**: A policy describes the behavior of the agent in a state. It is a mapping from the state of the environment to the action to be performed on that state. The agent tries to learn an optimal policy to choose the best action for each state. A policy in the game of chess can be described as a mapping from the position of chess pieces to the optimal action that can lead to a victory for the agent.

- **Value Function**: A value of a state describes the $goodness$ of the state. The value of a state is the total amount of reward an agent can expect to collect in the long term, starting from that state. The value of each of the possible state, s', that can be reached from a given state, s, determines the action chosen on that state. The value of a state in the game of chess describes how likely it is that the agent can win the game. The value of a state where the agent is about to lose will be very low as compared to the initial state of the chessboard.

- **Model**: A model of an environment is an approximation of how the environment behaves in any situation. It is an optional element in reinforcement learning and is used

to create *model-based* methods to train the agent.

2.3.2 Reinforcement Learning : Overview

To learn about the environment, the RL agent starts with exploring the environment. To explore, the agent performs random actions and observes the reward it gets from the environment. Based on such experiences, the agent learns the mapping between the situation and the optimal action to choose. Choosing the optimal action based on the experiences is known as exploitation. One of the major challenges in reinforcement learning is the trade-off between exploration and exploitation, i.e. when to stop exploring and start exploiting the information. To maximize its cumulative reward, the agent has to perform the optimal action each time and to learn this optimal mapping it has to explore more. If the agent doesn't explore, it won't be able to learn meaningful information and thus will fail to reach its goal. On the other hand, if the agent performs too much exploration, it won't be able to leverage the information learned from past experiences.

The other major challenge in reinforcement learning is to perform actions keeping future rewards in mind. The goal of the agent is to maximize its cumulative reward, and it doesn't imply to choose an action with the highest immediate reward. An immediate positive reward doesn't necessarily translate to getting closer to the goal state with the maximum cumulative reward.

To explain the working of an agent, we take an example of robot maneuvering. In this environment, the robot is tasked to reach a target location on the floor from a given start location. The floor (environment) can have obstacles which are not known to the agent a priori. The agent is allowed to move forward, backward, left, or right which constitutes the set of possible actions. The agent gets rewarded only when it reaches the target location and is penalized every time it hits an obstacle. At any block, the agent can see only one block in each direction, and this is the state of the environment for the agent. The agent decides the direction of its movement based on the state.

To learn to maneuver in this environment, the agent starts to explore the environment. In the beginning, the value of each state is the same, and the policy is uniform, i.e., each action is equally likely to happen, as shown in figure 2.5a and 2.5c. Initially, it moves randomly and collects data. Figure 2.4 shows one such run of the agent. The agent starts exploring the possible paths from its present location. Every time the agent hits an obstacle, it learns not to take that action on that state. When the agent reaches the target location, it gets a reward that incentivizes it to take actions such that it can reach this state. After multiple iterations, it learns the path from the start location to the target location, avoiding all obstacles. Figure

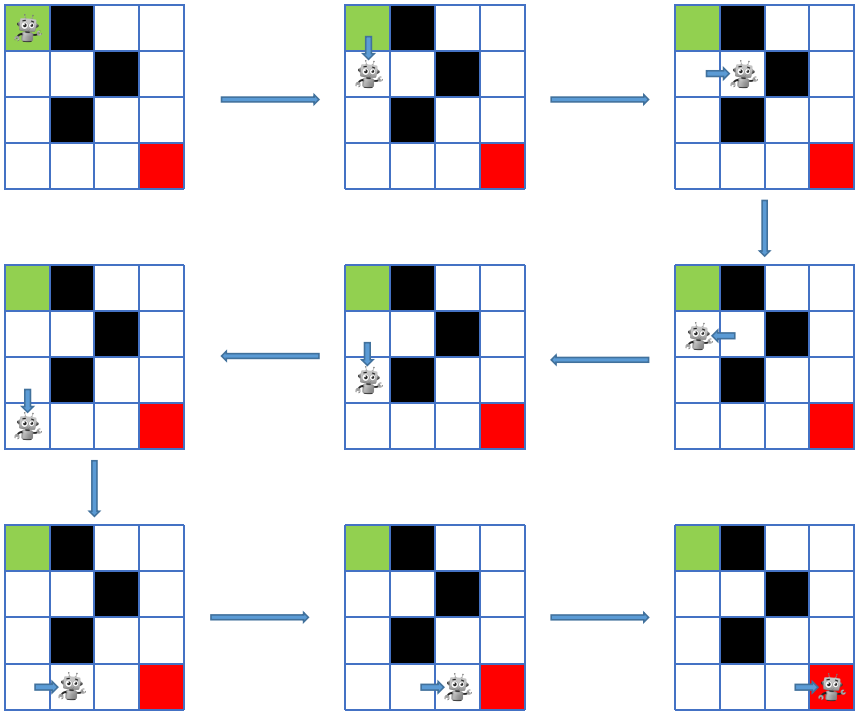


Figure 2.4: Reinforcement learning - Robot maneuvering environment

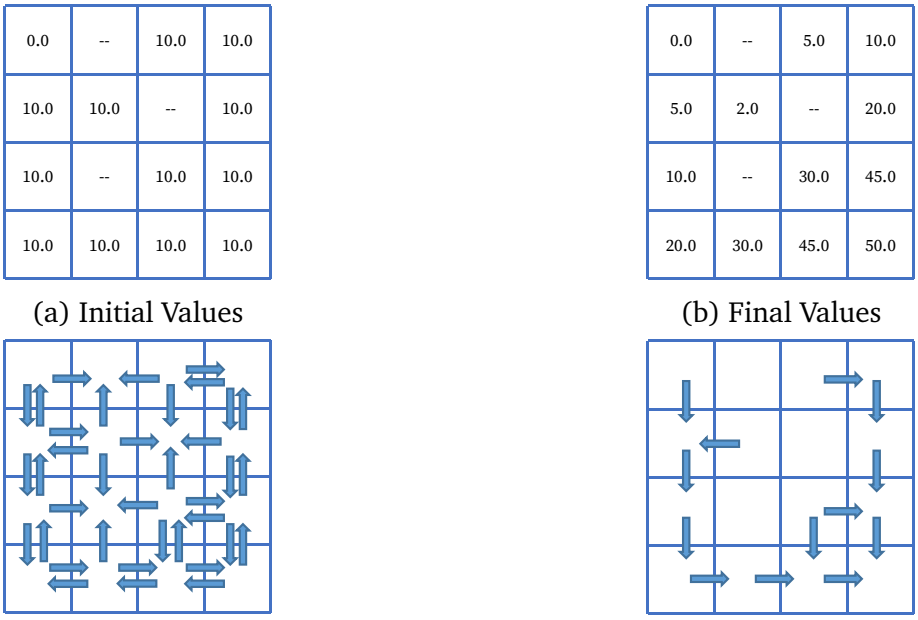
Each block represents a possible location for the agent. Special blocks are color-coded as: Green block: Start location, Red block: Target location, and Black block: Obstacle. The goal of the agent is to reach the target location with the highest cumulative reward. In this context, the agent receives a reward only when it reaches the target location and receives a penalty when it hits an obstacle.

2.5b show the value for each state and figure 2.5d shows the policy learned for a trained agent.

2.3.3 Deep Reinforcement Learning

The above example shows a simple environment and how an agent learns to take the optimal path for any given position. In this example, the number of possible states is small (just 13) and maintaining values for each state-action pair is reasonably straightforward. However, for real-world problems, the state space is enormous and storing and maintaining values for each state-action pair is not a feasible option. To mitigate such problems and increase the practicality of RL algorithms, Mnih et al. [19] came up with deep reinforcement learning.

For a trained agent, each state-action pair is uniquely mapped to a specific value. Using deep reinforcement learning, we try to find this function or an approximation of the function.



(a) Initial Values (b) Final Values

(c) Initial Policy (d) Final Policy

Figure 2.5: Reinforcement learning - Value and Policy

Neural networks are generally employed as a function approximator for this purpose. These networks take a representation of the state in vector form and give the optimal action for that state. These networks have successfully learnt to play Atari games [19].

Over the years, there have been many efforts to decrease training time and increase the stability and robustness of the training process. Describing each algorithm is beyond the scope of the thesis. We have compared some of the popular algorithms in section 4.2.

Chapter 3

Design

In this chapter, we discuss our approach and give details of our implementation choices. First we provide insight on why we chose reinforcement learning for our purpose. We move on to a high-level overview of the various components of our approach. Then we explain the various components of reinforcement learning in our environment. We end the chapter with the description of changes made in the fuzzing algorithm.

3.1 Why Reinforcement Learning

Machine learning has three main areas, supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the agent needs to have lots of definitive instances. Definitive instance means that the best action for a state is known apriori. Generally, an Oracle is present that tells the agent whether an action taken is correct or not. In our environment, there is no such Oracle present. Also, in the fuzzing environment, applying the same mutation multiple times on the same program input results in different rewards. Thus agents based on supervised learning are not useful.

On the other hand, in unsupervised learning, the agent doesn't need any such oracle for training. It can be easily argued that training an agent without any knowledge of how close it is from the target is a tough task. After each program execution, we have information about the closeness of the agent to the target location. It would be beneficial to use this information to train an agent.

This brings us to reinforcement learning. Reinforcement learning involves training an agent to accomplish a pre-defined goal. It tries to imitate humans by trial and error. The agent performs some action on the state of the environment and receives a scalar reward for the action. The agent interprets the reward as the *goodness* of its decision. The agent trains

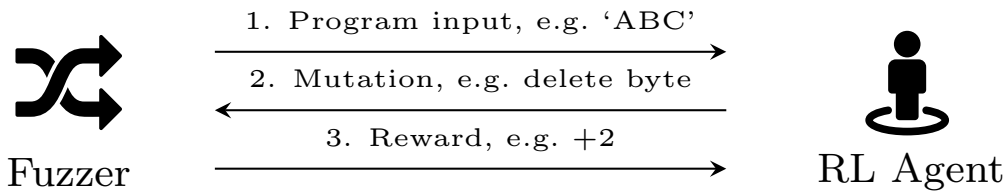


Figure 3.1: Block diagram of interaction between fuzzer and the RL agent

on these intermediate rewards and tries to maximize the cumulative reward for a sequence of actions. For directed testing, we visualize the agent trying to mutate (action) the program input (state) to reach the target location. Based on the change of distance from the target location, we can either reward or penalize the agent. We describe our approach in detail in the following sections.

3.2 Overview

Presently, any fuzzer chooses the mutations randomly. The choice of mutation is not dependent on the program input. Karamcheti et al. [26] have shown that not all mutations are beneficial for a given program input. In this thesis, we intend to find this relation between the program input and the optimal mutation. We employ reinforcement learning to train an agent to learn this relation.

Our approach integrates reinforcement learning in fuzzing. Figure 3.1 shows the interaction between the fuzzer and the RL agent. In step 1, the fuzzer sends the program input to the RL agent. The RL agent decides on a mutation based on past experiences and responds to the fuzzer in step 2. Based on the response from the agent, the fuzzer mutates the program input to generate a mutated program input. The fuzzer then executes the program with the mutated input. After getting the runtime information, the fuzzer sends the mutated input and the reward to the RL agent, termed as an experience as shown in step 3. Over time, based on these experiences, the RL agent tries to learn a better mutation strategy based on program inputs.

3.3 Distance Calculation

To measure the closeness of a seed input to the target location, we compute the distance from each basic-block to the target location. We start by generating the interprocedural control-flow graph (ICFG) from the program source code. The distance from a basic-block to the target location is calculated from the ICFG. Generating an ICFG from the program source

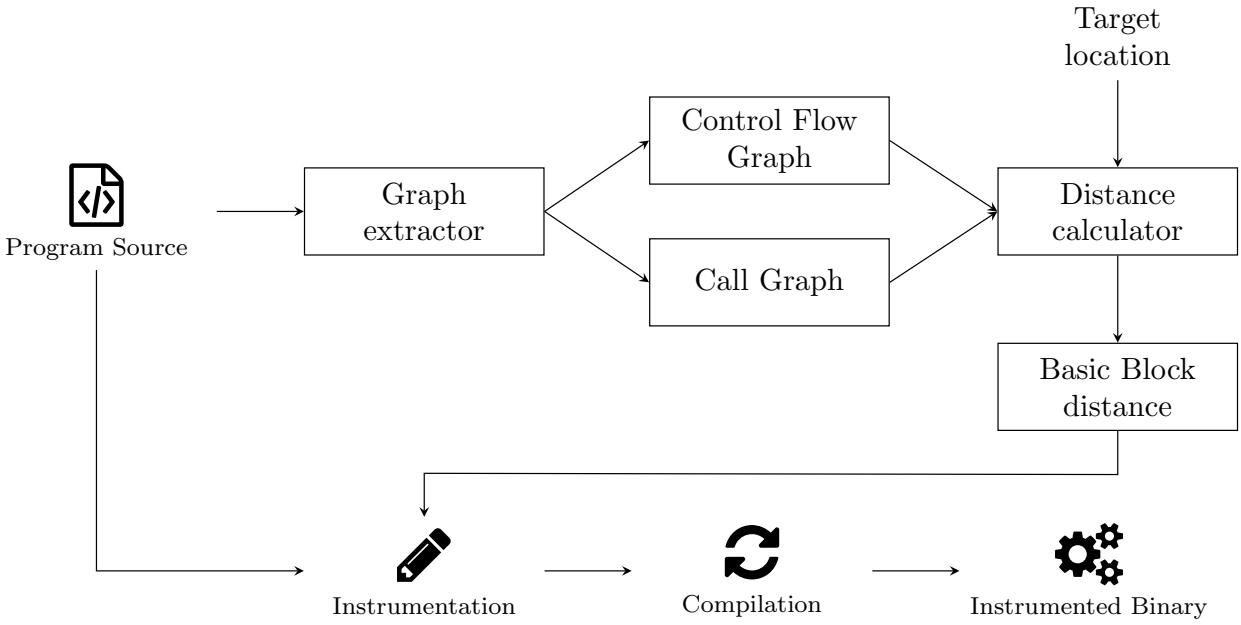


Figure 3.2: Block diagram of the modified compilation phase of AFL

is an arduous task, time-consuming, and often imprecise. To minimize the time complexity and to simplify the process, we calculate an approximation of this distance.

Figure 3.2 shows the changes in the compilation phase of the fuzzer. First, from the program source, we generate the control-flow graph (CFG) of each function and the callgraph (CG) of the whole program. These graphs are readily available in the LLVM compiler infrastructure. The target location is provided by the user during compilation. The basic-block that contains the target location, is referred to as the *target basic-block* (T_b) and the function containing the *target basic-block* is referred to as the *target function* (T_f). We calculate the *function level target distance* and *basic-block level distance* for each function and basic-block, respectively.

Function distance $d_f(n, n')$ between two functions n and n' is defined as the shortest distance between the functions in the callgraph. The distance is calculated as the number of edges in the path. *Function c distance* D_f for a function n is defined as the harmonic mean of the *function distance* between the function n and the reachable target function T_f.

$$D_f(n, T_f) = \begin{cases} undefined & \text{if no path from } f \text{ to } T_f \\ d & \text{otherwise} \end{cases} \tag{3.1}$$

where d is the harmonic mean of the *function distance* from function f to all reachable target

functions T_f.

Note that the *function level target distance* is calculated for multiple target functions. In our experiments, we have only one target function, so the *function level target distance* for a function is same as its *function distance* to the target function. This is useful when there are multiple target functions.

Basic-block distance, $d_{bb}(b_1, b_2)$, between two basic-blocks b_1 and b_2 that belong to the same function f, is defined as the length of the shortest path from b_1 to b_2 in the CFG of the function f. *Basic-block level target distance*, $D_{bb}(m, T_b)$, on the other hand, is defined for each basic block as the distance from a basic-block m to the *target basic-block*, (T_b). It can span across function calls. It is calculated as the harmonic mean of the *basic-block distance* to any basic-block that calls a function towards the target basic-block.

$$D_{bb}(m, T_b) = \begin{cases} 0 & \text{if } m \in T_b \\ c \cdot \min\limits_{n \in N(m)}(D_f(n, T_f)) & \text{if } m \in T \\ \left[\sum\limits_{t \in T}(D_{bb}(m,t) + D_{bb}(t, T_b))^{-1}\right]^{-1} & \text{otherwise} \end{cases} \tag{3.2}$$

where $N(m)$ is the set of functions called by the basic-block m, and T is the set of basic-blocks in the function containing m that has a function call. Here c has a fixed value of 10 and is an approximation the function-level distance.

Similar to *function level target distance*, *basic-block level target distance* is useful when there are multiple target basic blocks. Note that the distance calculation is the same as the one used by Boehme et al. in AFLGo [12].

3.4 RL components

3.4.1 State

A fuzzer interacts with the program under test by mutating the input to the program. We model the sequence of bytes of the program input as the state. Thus, the state is represented as

$$S = t_0 t_1 t_2 ... t_{n-1}$$

Here t_i represents the i^{th} byte in the program input and n is the length of the program input. As the length of each program input is variable, we cap the length of each program input to some threshold. For program inputs with length less than the threshold, we pad it with zeros.

For program inputs with length more than the threshold, we truncate the input accordingly.

3.4.2 Action

We define the set of mutations applied by the fuzzer on each program input as the action set for the agent. The mutations applied by the fuzzer is described below:

- **Bit flip**: Flip a bit at any location in the program input.
- **Overwrite with interesting values**: Overwrite a byte with some interesting values like 0, 1, 32768, -1. The values range from 1 byte to 4 bytes.
- **Add to a byte**: Add a random integer to a byte, word, or double word.
- **Subtract from a byte**: Subtract a random integer from a byte, word, or double word.
- **Random overwrite a byte**: Set a random byte to a random value.
- **Delete bytes**: Delete bytes from a random location in the input. The fuzzer decides the block length at runtime.
- **Insert bytes**: Insert bytes at a random location in the input. The inserted block is either a clone of the existing input or a block of constant bytes. The fuzzer decides the block length at runtime.
- **Overwrite bytes**: Overwrite bytes from a random location in the input to another location in the input. The fuzzer decides the block length at runtime.

3.4.3 Reward Function

The reward for actions by the agent is calculated based on the closeness to the target location. The agent receives a reward when the mutation (action) suggested by it results in a new program input (state) that is closer to the target location than the previous program input (old state). Similarly, if the new program input after applying the mutation moves away from the target location, the agent is penalized.

Since the agent is sensitive towards the reward function, we have tested our approach with various reward functions. We describe each of them as follows:

- **Average Distance**: In this method, we first calculate the distance of each basic-block to the target location. The distance is calculated from the interprocedural control-flow graph (ICFG) of the program as described in section 3.3. Note that the distances are

calculated for only those basic-blocks which have a path to the target location in the ICFG.

The seed distance is then calculated as the mean of the distances of the executed basic-block to the target location.

$$dist_{seed} = \frac{\sum_{m \in N(seed)} D_{bb}(m, T_b)}{|N(seed)|}$$

where $N(seed)$ is the set of basic blocks executed by the $seed$ and have a path to the *target basic-block*, (T_b). The reward is calculated as:

$$reward = dist_{org} - dist_{mut}$$

where $dist_{org}$ is the seed distance of the original seed ($seed$) and $dist_{mut}$ is the seed distance of the mutated seed ($seed'$).

- **Least Distance**: This method calculates the basic-block level target distance for each seed as described in the previous method. To calculate the seed distance for each seed input, we find the closest basic-block to the target location that was executed by the seed input. The distance from this basic-block to the target location is the seed distance for that seed input.

$$dist_{seed} = \min_{m \in N(seed)} (D_{bb}(m, T_b))$$

Here $N(seed)$ is the set of basic blocks executed by the $seed$ and have a path to the *target basic-block*, (T_b). The reward is calculated similarly to the previous method.

$$reward = dist_{org} - dist_{mut}$$

where $dist_{org}$ is the seed distance of the original seed ($seed$) and $dist_{mut}$ is the seed distance of the mutated seed ($seed'$).

- **Basic-Block Hit Count**: In this method, we count the basic-blocks that were executed by the seed input and can reach the target location. The idea behind this technique is that a seed input which is closer to the target location would execute a higher number of such interesting basic-blocks compared to a seed input which is farther away from the target location.

Note that we count each basic-block only once. This is done to avoid counting the interesting basic-blocks inside loop multiple times. So the hit count for a seed input is:

$$bbhc_{seed} = |N(seed)|$$

Here N(seed) is the set of basic blocks executed by the $seed$ and have a path to the *target basic-block*, (T_b). The reward is calculated as:

$$reward = bbhc_{mut} - bbhc_{org}$$

where $bbhc_{org}$ is the hit count of the original input and $bbhc_{mut}$ is the hit count of the mutated seed.

We have evaluated each of the technique mentioned above, and the results are presented in section 4.3.

3.5 Fuzzing algorithm changes

To integrate reinforcement learning into fuzzing, we made some changes in the fuzzing algorithm. We highlight these changes in Algorithm 2. The function $select\_mutation()$ is modified to receive a seed input ($seed$) as a parameter. It sends the $seed$ to the RL agent, as shown in figure 3.1 and waits for the response from the agent. This encapsulates both step 1 and 2 from figure 3.1. It then applies the mutation and checks for interesting behavior.

The helper function $GET\_REWARD()$ determines the reward for each mutation. The reward is determined based on the difference of the distance to the target location in the interprocedural control flow graph, as already discussed in subsection 3.4.3. Rewards are normalized to the range [-1,1] for stability in gradient updates. $send\_experience()$ sends the experience to the RL agent for training and completes the step 3 in figure 3.1.

Algorithm 2: American Fuzzy Lop - Modified Fuzzing Phase Algorithm

Input: Program $\mathcal{P}$, Program arguments $\mathcal{A}$, Seed Inputs $\mathcal{S}$
Output: Interesting Seeds $\mathcal{Q}$

```
1  Function AFL_Fuzz_Loop(P, A, S)
2      Q ← φ
3      foreach seed ∈ S do
4          add_to_queue(Q, seed)
5      end foreach
6      while True do
7          seed ← select_seed(Q)
8          energy ← calculate_score(seed)
9          for i ← 1 to energy do
10             mutation ← select_mutation(seed)
11             seed' ← mutate_seed(seed, mutation)
12             status ← run_target(P, A, seed')
13             if status == INTERESTING then
14                 add_to_queue(Q, seed')
15             end if
16             reward = GET_REWARD(seed, seed')
17             send_experience(seed, mutation, reward, seed')
18         end for
19     end while
20     return Q
21 end Function
22
23 Function GET_REWARD(seed, seed')
24     distance = get_distance(seed)
25     distance' = get_distance(seed')
26     reward = distance − distance'
27     reward = normalize(reward)
28     return reward
29 end Function
```

Chapter 4

Evaluation and Lessons Learned

In this chapter, we describe and evaluate our approach across various synthetic programs. We tried several algorithms and strategies, and we assess each of them separately and summarise our findings. We start by comparing different training mechanisms for the agent. It is followed by a comparison of different reinforcement learning algorithms. Finally, we evaluate the different reward functions on some synthetic and real-world programs.

4.1 Training mechanism

Any reinforcement learning (RL) agent needs to learn about the environment to make informed decisions. To acquire this information, the agent starts by gathering experiences. These experiences are the accumulation of the continuous interaction between the agent and the environment. Each experience *E* is a tuple of 5 elements.

$$E = \{s, a, r, s', d\}$$

where,
s : State of the environment, $s \in \mathcal{S}$, finite set of states.
a : Action selected for the state s, $a \in \mathcal{A}$, finite set of actions.
r : Reward for the action a selected for state s, $r \in \mathbb{R}$
s' : New state of the environment after executing the action a on state s, $s' \in \mathcal{S}$
d : Has the agent reached its goal, $d \in \{0, 1\}$

The agent can only choose the action a for any state s. Initially, the agent starts by picking random actions. The agent observes the reward for each of these random actions. The agent

repeatedly performs these steps to improve its knowledge of the environment and chooses the best action for any given state. This kind of learning by continuously interacting with the environment is known as *online learning*.

Online learning is the most common approach for training agents. Although it is known to work for a variety of tasks, the agent converges slowly, and thus the training takes a long time. The agent has to wait for the reward and the new state after each interaction with the environment. The continuous interaction with the environment adds considerable delays in training time.

To navigate around the problem of waiting for the response from the environment, we came up with an approach of training the agent with saved experiences. We term it as *offline learning*. In *offline learning*, the agent doesn't interact with the environment during the training phase. The idea is to let the agent train on saved experiences without the need for continuous interaction with the environment. *Offline learning* can reduce the training time for the agent by a considerable margin.

4.1.1 Environment

We start with a simple task to compare the performance of *online* and *offline learning*. In this task, the goal of the agent is to mutate any given character string such that it transforms into a different target character string. The target character string is not known to the agent a priori. The agent is allowed to mutate the string in three ways to transform a given string to the target string.

- Insert character (1): This mutation inserts a random character at a random location in the source string.
- Delete character (2): This mutation deletes a character from a random location in the source string.
- Overwrite character (3): This mutation overwrites a character in the source string at a random location with a random character.

In this task, we model the character string as the agent state. The agent gets a reward when the mutation on the source string leads it closer to the target string. To measure the closeness of two strings, we use Levenshtein distance [27]. Levenshtein distance or edit distance is a popular technique to measure differences between two strings. It measures the minimum number of single-character edits required to transform a given source string to a target string.

Example 4.1.1. *Example showing the action and reward mechanism for the environment.*
Source string : 'abc'
Target string : 'fuzz'

Step	Source String	Action	Mutated String	Reward	Distance
1	abc	3	abb	0	4
2	abb	3	fbb	1	3
3	fbb	1	fubb	1	2
4	fubb	2	ubb	-1	3
5	ubb	3	fbb	0	3
...	...	...	...	...	...
80	fuz	1	fuzz	100	0

Table 4.1: Steps showing each interaction of the agent with the environment.

An experience for the agent is a 5-tuple $\{s, a, r, s', d\}$:
Step 1 : **{'abc', 3, 0, 'abb', 0}**
Step 2 : **{'abb', 3, 1, 'fbb', 0}**
...
Step 80 : **{'fuz', 1, 100, 'fuzz', 1}**

Rewards for each action is determined as follows:

- **Positive (+1)**: The mutation decreases the edit distance between the mutated and target string, as shown in step 2 and 3 in example 4.1.1.
- **Negative (-1)**: The mutation increases the edit distance between the mutated and target string, as shown in step 4 in example 4.1.1.
- **Null (0)**: The mutation has no effect on the difference of edit distance between the mutated and target string, as shown in step 1 and 5 in example 4.1.1.
- **Completion (+100)**: The mutation successfully converted the string to the target string as shown in step 80 in example 4.1.1.

A high reward for completion of task acts as an incentive for the agent to complete the task. Also, to increase the robustness of the agent, we start each episode with a different source string.

Thus, the task can be pictured as an agent trying to modify a given random string to a fixed target string by continuously mutating the source string. The agent has to learn the best mutation for any given state and successfully steer away from any bad mutations. For evaluation, we compare three agents, each based on a different learning mechanism.

These are described as follows:

- **Random Agent**: Selects a random action for any given string. The action chosen is entirely oblivious of the state.
- ***Online* learning based RL agent**: Selects an action based on the string. It continuously interacts with the environment and learns from its own experiences.
- ***Offline* learning based RL agent**: Selects an action based on the string. It doesn't interact with the environment and learns from the experiences of a different agent. For our experiment, we have trained this agent using the experiences of the random agent.

We have used Double Duelling Deep Q Networks (DDQN) algorithm [22] to train both the RL agents. Due to timing constraints, in these experiments, we have also limited the maximum size of the string to 32 bytes.

4.1.2 Experiment Configuration

We implemented our techniques in Tensorflow [28] using an open-source implementation of DDQN [29]. We modified the implementation to adapt to our environment. In particular, we have used three 1-dimensional Convolution Neural Network (CNN) layers and a fully connected layer to encode the state. Each convolution layer has a kernel of size 9 and a stride of length 3 and uses ReLU as their activation function. The final fully connected layer uses sigmoid as its activation function. Two dropout layers, each with a dropout rate of 25%, are also added after the final convolution layer and the fully connected layer. As the input length can vary, we have fixed the size of the agent state to 200 bytes and padded the input with 0 whenever necessary. Please note that this limit differs from the string length limit that is capped at 32 bytes for experiments.

All experiments are carried out on a system which has a pair of Intel Xeon Gold 2.7GHz processors totaling 72 cores and 256 GB of memory running Ubuntu 16.04. The agent was also provided access to an NVIDIA 1080Ti 11GB GPU to accelerate the training process. For *offline learning*, we generated experiences from the random agent. We ran the random agent for 3 hours and collected 3.6 million experiences for training. We trained each agent for 7 hours and saved the trained model after every 15 minutes. Each episode length is capped at 5000 mutations, and the agent starts a new episode with a new random string.

Approach	Total Episodes for training
Offline	8024
Online	2022

Table 4.2: Total number of episode runs for training

4.1.3 Evaluation

Any agent that claims to learn this simple task should be able to transform any given string to the target string. We selected a set of 100 random strings for testing. We tested each saved model from both the agents against this test suite. An episode run is said to be successful if the agent was able to transform the given source string to the target string within the limit of the number of mutations. We ran each experiment 5 times, and plot the average of all the runs as shown below. We compare both of our RL agents against the baseline agent.

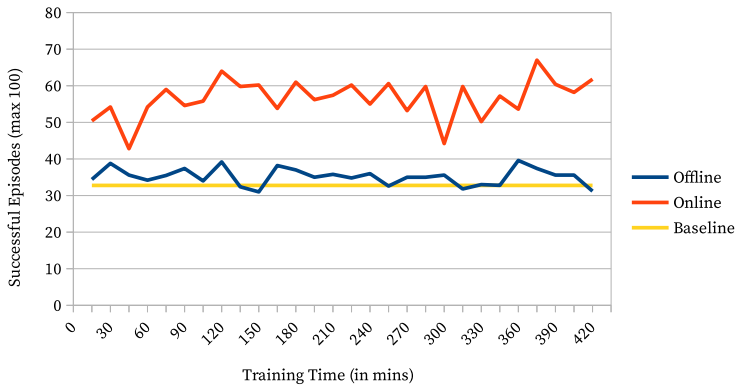


Figure 4.1: Comparison of *Offline* and *Online* training w.r.t. baseline random mutation

This experiment shows that *online learning* can learn the appropriate mutation to be applied to each state. Although it was not able to successfully mutate all the random strings from the test suite to the target string, it was able to increase the efficiency of the baseline agent by 64%. On the other hand, our third agent that employed *offline learning* was not able to acquire any relevant information and performed similar to the random agent. It is contrary to our expectations.

We suspect that as the agent was learning from the experiences of the random agent, it was limited to those experiences. The agent was unable to verify its choice of action. In the case of our agent based on *online learning*, it repeatedly learned from its own experiences. It allowed the agent to learn from any mistakes made during training. Also, as training con-

tinued, our *online learning* based agent was able to gather more experiences with positive rewards, which could have acted as positive feedback for the agent. For the agent based on *offline learning*, there was no such positive feedback. It relied on the experiences of the random agent that had no such feedback mechanism.

Lessons Learned
• Offline learning is not able to learn any relevant information from stored experiences.
• Online learning, which is very slow, can figure out the appropriate mutation strategy.

4.2 Comparison of different RL algorithms

In the past few years, there has been a multitude of work in the area of reinforcement learning. There have been many algorithms developed to decrease the training time for the agent. The performance of these algorithms was mostly tested in game environments. There has been no prior work which explores different RL algorithms in software testing. In this section, we compare some of the popular training techniques used in reinforcement learning to choose the best algorithm to train our fuzzer.

4.2.1 Training Algorithms

- **Deep Q-Network (DQN)** : Q-learning is one of the basic algorithms to train an RL agent. In the most straightforward implementation, the agent maintains a table where the rows represent the state, and the columns represent the actions. The agent calculates the value, called Q-value, for each such state-action pair from the past experiences. Tables work fine for small environments where the state space is small, but it doesn't scale for larger environment with large state space. For handling such environments, neural networks are used as a function approximator. The network learns the mapping of the state representation to its corresponding Q-values. Neural networks help to calculate the Q-values for each state-action pair without the need to store it in a table.

- **Double Duelling Deep Q Network (DDQN)** : Although DQN has enabled to use reinforcement learning for a large number of problems, it faces stability issues and converges very slowly. Various enhancements have been made to work around these problems. Some of these enhancements include adding convolution layers to gather only relevant information from the state and using experience replay to increase the robustness of the learning process. Double DQN uses a second network to reduce overestimation, train faster and reliably. Duelling DQN also uses two networks, but to compute two

different functions (advantage and value) and combines them to get the Q-value for a state. Calculating two different functions, and combining them later helps in generating robust estimates of the state value. Double Duelling DQN combines all these techniques for the best possible results.

- **Asynchronous Advantage Actor-Critic Network (A3C)** : A3C is the most popular and the go-to algorithm for reinforcement learning tasks. The main differentiating factor is the employment of multiple agents which learn the task separately and periodically update a global network. It also uses convolution layers, experience replay, and utilizes two networks to learn the value function and the policy separately.

4.2.2 Environment and Configuration

We have used the same environment from the previous section to evaluate the above mentioned reinforcement learning algorithms. For DQN, we have used a single neural network layer as a function approximator. For DDQN, we have used the same configuration from the previous experiment. For A3C, we used an open-source implementation [29]. In particular, we modified the state representation as described in the previous experiment.

4.2.3 Evaluation

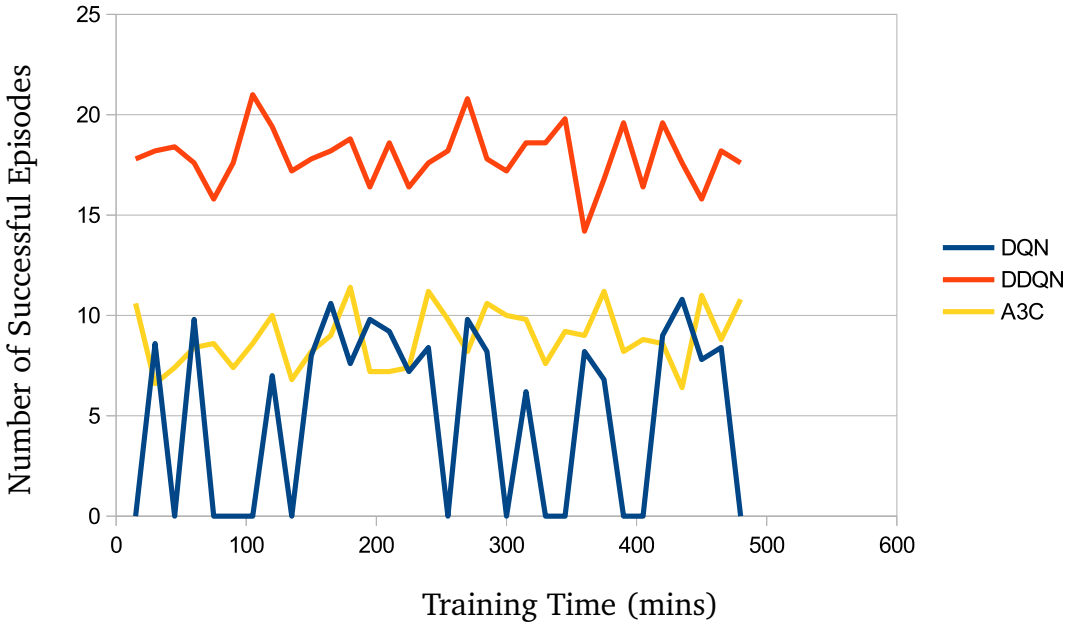


Figure 4.2: Comparison of different RL algorithms training on the same task.

Algorithm	Total Episodes for training
DQN	2082
DDQN	2241
A3C	3548

Table 4.3: Total number of episode runs for training

We trained the three agents for 8 hours and saved the model after every 15 mins. Each episode was capped at 5000 mutations, and every agent starts a new episode with a new random string. For testing, we generated 30 random strings of different length. We tested each saved model against the test set of strings. Each experiment ran for 5 times, and the average is plotted and shown in figure 4.2.

From this simple experiment, we observe that not all agents based on reinforcement learning algorithms can learn the appropriate mutation strategy. The first agent, based on Deep Q-Network (DQN), is not able to perform consistently over time. In our environment, the rewards are stochastic, i.e., applying the same mutation on the same state may result in different rewards. As the agent is highly sensitive to these rewards, getting negative rewards for the same actions that previously gave positive rewards destabilizes the algorithm. Our experiment confirms this drawback.

The second agent, based in Double Duelling Deep Q-Network (DDQN), is performing consistently. It has learned the appropriate mutation strategy, and unlike DQN based agent, it is not highly sensitive to immediate rewards. We credit the stability of the algorithm to the use of experience replay during the training phase. Experience replay is used to soften the sensitiveness of the agent towards immediate rewards.

The third agent, based on Asynchronous Advantage Actor-Critic Network (A3C), is not able to perform better than DDQN based agent. Although A3C is known to perform better than the DDQN in a variety of tasks, it was not able to perform any better than the random agent. Like DDQN, due to the employment of experience replay during the training phase, the network was stable.

Lesson Learned

- DDQN based agent outperforms both DQN and A3C based agents.

4.3 Synthetic Benchmarks

In this section, we evaluate our approach with the three reward functions, as described in section 3.4.3 on some synthetic programs. We choose to first test our approach on some small and simple programs to validate the feasibility of our approach. In each example, we train three agents, each based on one such reward function, and compare its performance with AFL and AFLGo. The three agents are labeled as **Average**, **Minimum**, and **Hit Count** corresponding to each reward function as described in 3.4.3. Since our aim is to reduce the number of mutations applied to reach the target location, we use it as a performance metric. We also compare the time taken by each agent to reach the target location.

4.3.1 Experimental Configuration and Plot

In each experiment, the fuzzer starts with an empty string as the seed input. We use DDQN as the choice of algorithm for training the agent. We used the same configuration as used in 4.1.2 for state representation. As the input to the program varies in length, we capped the length of the program input in each case which is mentioned in each test case. We trained each agent for 16 hours and saved the model every 1 hour. For testing, we used the last saved model.

Each experiment was repeated 10 times for statistical significance. Performance of each agent is plotted in the standard box-and-whiskers plot. Each box is bounded by the 1st and 3rd quartile values. The centre line in each box shows the median value. The lines extending from box on each side show the minimum and maximum values. Any outlier is shown as a dot outside the range of the whiskers.

4.3.2 Program - 1: Consecutive same characters

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <assert.h>
4
5  int main (int argc, char *argv[])
6  {
7      char str[30];
8
9      scanf("%30s", str);
10     str[29] = '\0';
11
12     if(str[0] == 'A')
```

```
13      if(str[1] == 'A')
14       if(str[2] == 'A')
15        if(str[3] == 'A')
16         if(str[4] == 'A')
17          if(str[5] == 'A')
18           if(str[6] == 'A')
19            if(str[7] == 'A')
20             if(str[8] == 'A')
21              if(str[9] == 'A')
22               if(str[10] == 'A')
23                if(str[11] == 'A')
24                 if(str[12] == 'A')
25                  if(str[13] == 'A')
26                   if(str[14] == 'A')
27                    if(str[15] == 'A')
28                     if(str[16] == 'A')
29                      if(str[17] == 'A')
30                       if(str[18] == 'A')
31                        if(str[19] == 'A')
32                         if(str[20] == 'A')
33                          if(str[21] == 'A')
34                           if(str[22] == 'A')
35                            if(str[23] == 'A')
36                             if(str[24] == 'A')
37                              if(str[25] == 'A')
38                               if(str[26] == 'A')
39                                if(str[27] == 'A')
40                                 if(str[28] == 'A'){
41                                   assert(str[2] == 'B');
42                                  }
43    return 0;
44 }
```

We start with a very simple program that accepts a string from the user and checks if the input string starts with 29 consecutive 'A' and aborts the program if it is the case. In this example, as the fuzzer finds an input that reaches closer to the target location, the mutation involving overwriting bytes should be preferred over any other mutation. Our agent should be able to learn this fact from experience and should be able to reach the target location in a fewer number of mutations. The target location in this program is line 41 and the program input length is capped at 32 bytes.

Figure 4.3 shows the number of mutations, and figure 4.4 shows the time taken by each

fuzzer to find an input that reaches the target location. All the agents were able to find the crashing input in fewer mutations than the baseline. On average, our approach reduced the number of mutations by a factor of 2.8x w.r.t. AFL and 1.3x w.r.t. AFLGo. However, in terms of time taken, our approach was 2-3x slower than AFL and 4-5x slower than AFLGo to reach the target location. The fuzzer had to wait for the agent's response before each mutation, and as a result, it got slowed down.

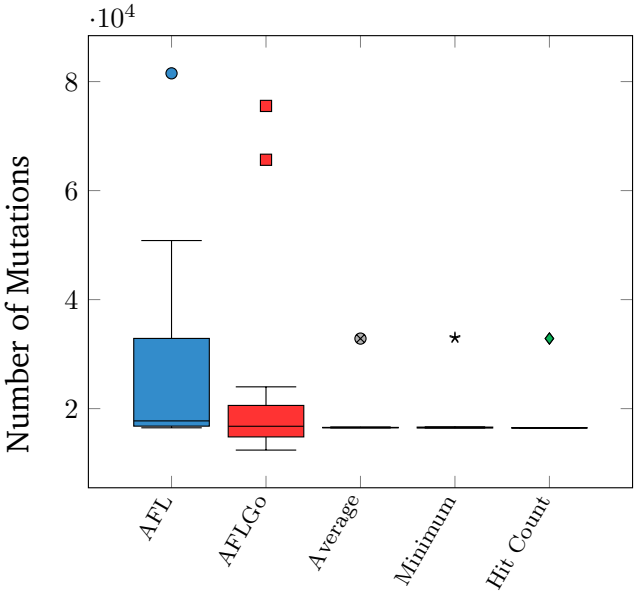


Figure 4.3: Program-1: Mutations applied by each agent to reach the target location.

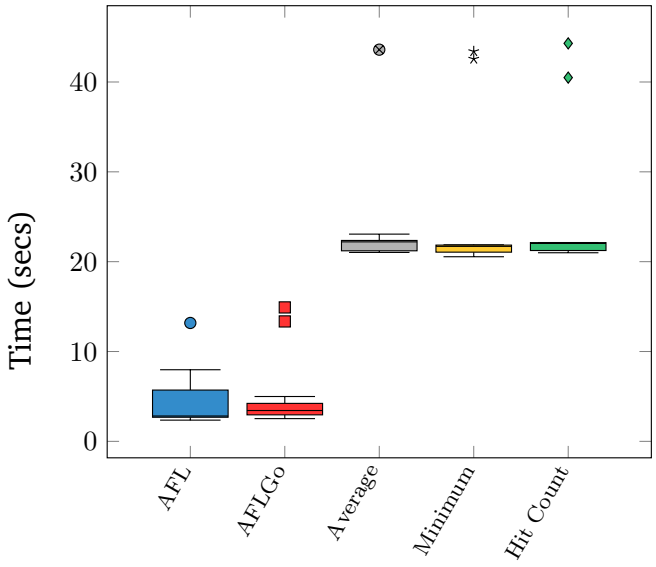


Figure 4.4: Program-1: Time taken by each agent to find an input that reaches the target location

4.3.3 Program - 2: Arithmetic operations

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <assert.h>
5
6  long int convert_to_int(char *s)
7  {
8      long int i = strtol(s, NULL, 10);
9      if(errno == ERANGE || i == 0)
10         return -1;
11     return i;
12 }
13
14 int main()
15 {
16     char *str = (char *) malloc(11);
17     scanf("%10s", str);
18     str[10] = '\0';
19     if(str[0] == 'f')
20       if(str[1] == 'u')
21         if(str[2] == 'z')
```

```
22              if(str[3] == 'z')
23              {
24                  long int r = convert_to_int(str+4);
25                  if(r != -1 )
26                      if(r % 25 == 0)
27                          assert(r%25 == 1);
28              }
29      return 0;
30  }
```

In the next example, we introduce arithmetic operations in the program. Unlike the previous example, there is no clear choice of mutation for all inputs, thus making the agent's task arduous. The program accepts a string from the user and crashes if the string starts with the literal "fuzz" followed by an integer that is a multiple of 25. The program input was capped at 16 bytes.

Figure 4.5 shows the number of mutations, and figure 4.6 shows the time taken by each fuzzer to find an input that reaches the target location. In this example, only one agent (**Hit Count**) was able to match the performance with both the baseline fuzzers. It also has low variance compared to both AFL and AFLGo. However, it was 3.8x and 5.2x slower than AFL and AFLGo, respectively, to reach the target location. The other two agents, based on **Average** and **Minimum** distance, performed worse than the baselines both in terms of the number of mutations and time taken to reach the target location. This experiment shows that the agent's performance is highly dependent on the choice of reward function.

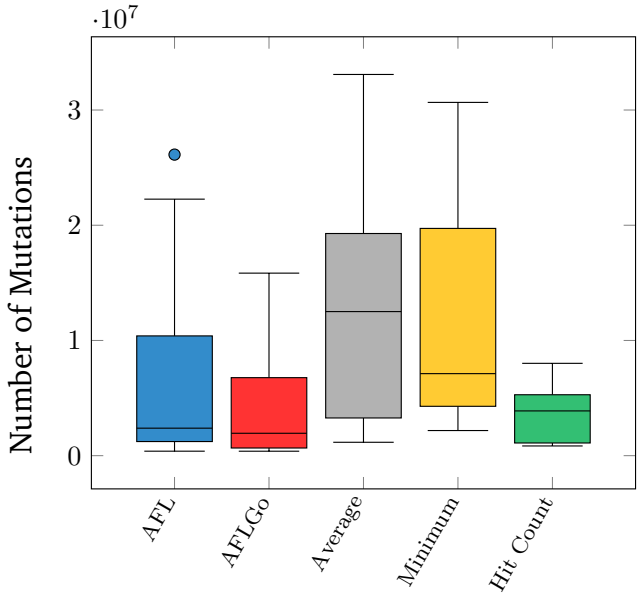


Figure 4.5: Program-2: Mutations applied by each agent to reach the target location

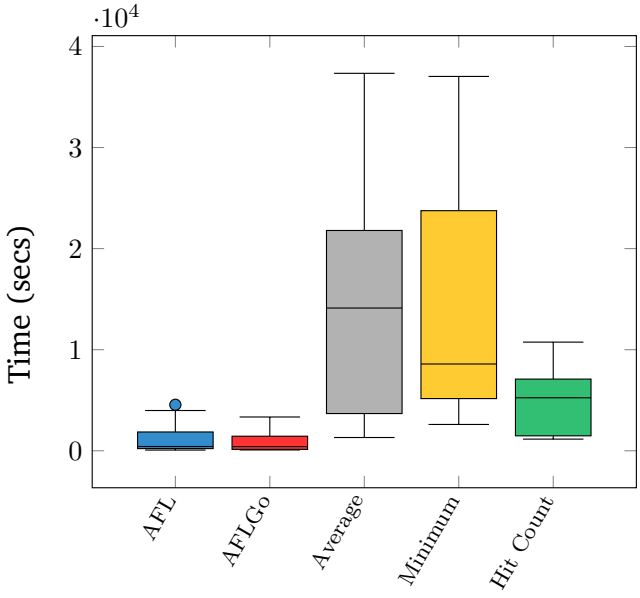


Figure 4.6: Program-2: Time taken by each agent to find an input that reaches the target location

4.3.4 Program - 3: Buffer Overflow

```
1 #include <stdio.h>
```

```
2  #include <assert.h>
3  #include <string.h>
4
5  int main(int argc, char **argv)
6  {
7    FILE *fp;
8    char line[40], str[20];
9    int read, len=40;
10   fp = fopen(argv[1], "r");
11
12   while (fgets(line, len, fp) != NULL) {
13       if(line[0] == '#')
14           continue;
15       strcpy(str,line);
16       break;
17   }
18   if (str[0] == 'f')
19     if (str[1] == 'u')
20       if (str[2] == 'z')
21         if(str[3] == 'z')
22           if (stelen(str) > 20)
23             assert(strlen(str) < 20);
24   return 0;
25 }
```

In this example, the program takes a filename as an argument, opens the file and reads it line by line. It copies the content of the first line to the string buffer *name*. The program ignores any line that starts with '#'. If the first valid line starts with "fuzz" and contains more than 20 characters, it fails the assertion at line 23. We set our target location at line 23 to find such strings and the file length was capped at 64 bytes.

From figures 4.7 and 4.8, we observe that AFL surprisingly outperformed all other fuzzers both in the number of mutations and time taken to reach the target location. It is contrary to our expectations as AFL is not "directed" towards the target location. However, our approach outperforms AFLGo, a directed fuzzer based on AFL, in terms of the number of mutations performed; but lags in the time it took to reach the target location. This example shows that fine-tuned heuristics are not always effective, and a data-driven approach can be useful.

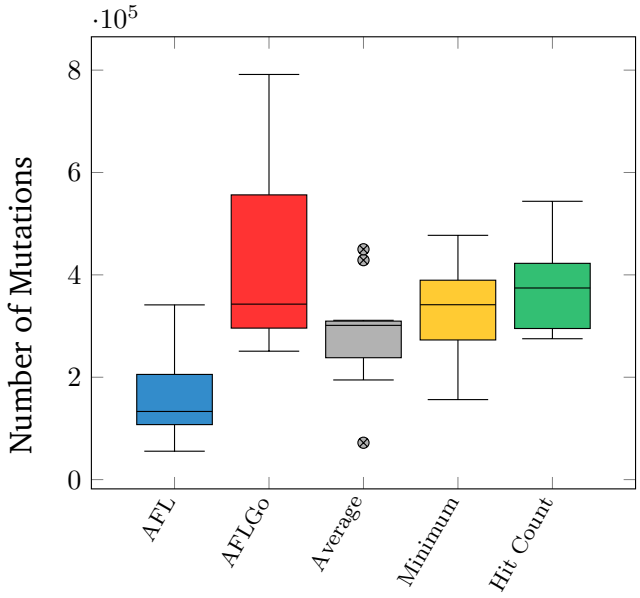


Figure 4.7: Program-3: Mutations applied by each agent to reach the target location

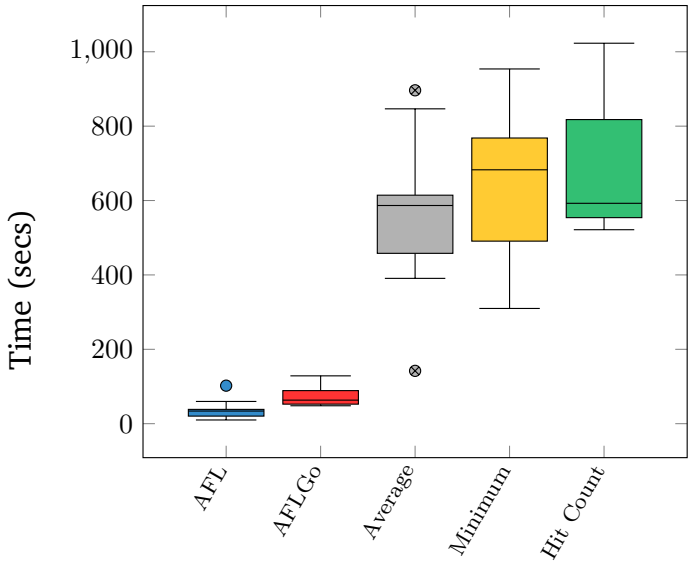


Figure 4.8: Program-3: Time taken by each agent to find an input that reaches the target location

4.3.5 Program - 4: Function Calls

```
1 #include <stdio.h>
```

```
2  #include <assert.h>
3  #include <string.h>
4
5  int calculate_half(char * str, int n)
6  {
7      int sum = 0;
8      for(int i = 0; i < n; i++)
9      {
10         sum += str[i];
11     }
12     return sum;
13 }
14
15 int check_palindrome(char *src, char * dest, int n)
16 {
17     if(strncmp(src, dest, n) == 0)
18         return -2;
19     for(int i = 0; i < n; i++)
20     {
21         if(src[i] != dest[n-i-1])
22             return -1;
23     }
24     return 0;
25 }
26
27 int calculate(char *str, int n)
28 {
29     if (n < 10)
30         return -1;
31     int v1, v2;
32     v1 = v2 = 0;
33     v1 = calculate_half(str, n/2);
34     if(v1 % 50 == 0)
35         v2 = calculate_half(str + n/2, n/2);
36     if(v1 == v2)
37         return check_palindrome(str, str + n/2, n/2);
38     else
39         return v1 - v2;
40 }
41
42 int main()
43 {
```

```
44      char str[30];
45      scanf("%30s",str);
46      if (!(str[0] >= 'a' && str[0] <= 'z') || strlen(str)%2 == 1)
47          return 0;
48
49      int r = calculate(str, strlen(str));
50      assert(r == 1);
51      return 0;
52  }
```

In the final synthetic example, we added more functions and constraints in the program. Here, the program takes a string from the user and checks if the input string is palindrome of even length. There are some prerequisites for the string that it needs to satisfy (line 51, 32, 37, and 41). The program crashes if it passes all the prerequisites and is also a palindrome. The target location for this program is set at line 56 and the input was capped at 32 bytes.

From the figure 4.9, we observe that the baseline fuzzers outperformed our apporach. Although the agent based on the minimum distance reward function performed better than the other two agents, it failed even to match the performance of AFL. Figure 4.10, which compares the time taken by each fuzzer, shows similar trends from the previous experiments.

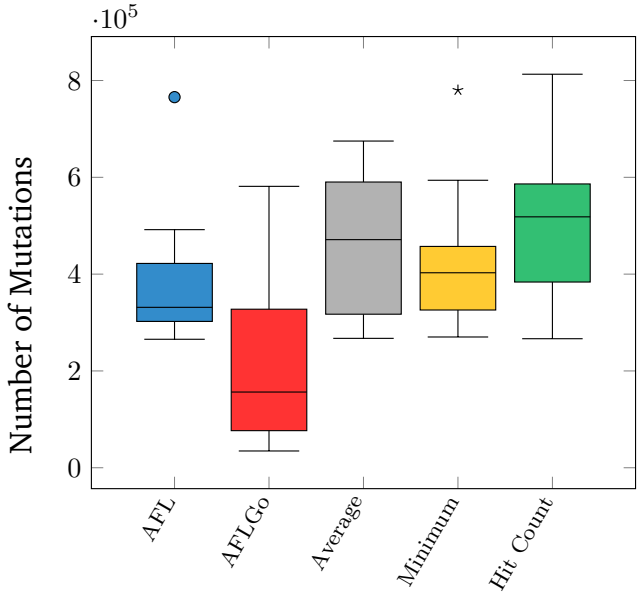


Figure 4.9: Program-4: Mutations applied by each agent to reach the target location

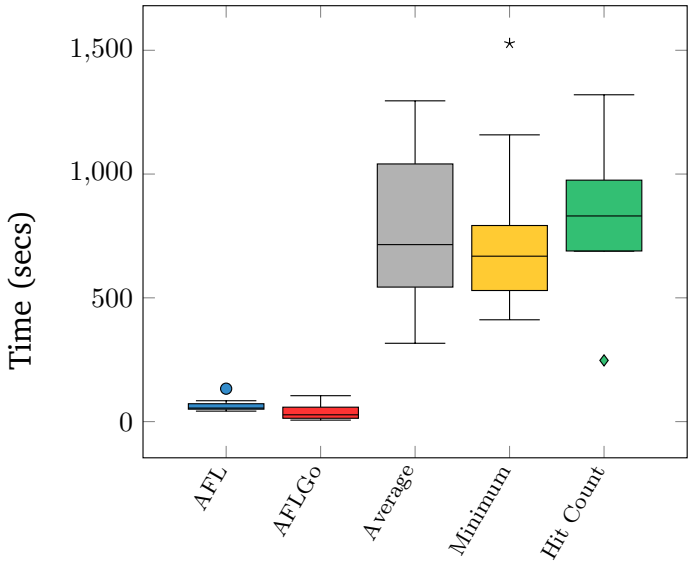


Figure 4.10: Program-4: Time taken by each agent to find an input that reaches the target location

Lessons Learned

- Agents are able to learn simple tasks but have mixed results in complex tasks.
- Agent's performance is heavily dependent on the reward function used.
- No one technique is the clear winner in all cases.

4.4 Real World Programs

We tried our approach on two real-world programs, namely *libxml2*, an XML parsing tool, and *binutils*, a *binutils* a collection of tools for creating and managing program binaries. Both these programs are reasonably large, with *libxml2* having more than 240k lines of code, and *binutils* has close to 1 million lines of code [1].

In *libxml2*, we selected a patch (commit id: ef709ce2 [2]). This commit fixes a bug [3] introduced by a patch for a fix of an earlier bug [4]. The first modified line in the patch was set as the target location for the fuzzer. The seed input for the fuzzer was a small, well-formed XML file. The length of the program input was capped at 4 kilobytes.

[1] calculated using Cloc, https://github.com/AlDanial/cloc
[2] https://gitlab.gnome.org/GNOME/libxml2/commit/ef709ce2
[3] https://bugzilla.gnome.org/show_bug.cgi?id=737840
[4] https://bugzilla.gnome.org/show_bug.cgi?id=724903

In *binutils*, we selected three vulnerabilities for testing. Each of these vulnerability has been assigned a CVE-ID and is listed in table 4.4 with its type. In each test, the target location for the fuzzer was set to the point of the crash. The fuzzer was seeded with an empty file, and the length of the program input was capped at 512 bytes.

CVE-ID	**Type of vulnerability**
CVE-2016-4487	Invalid Write
CVE-2016-4488	Invalid Write
CVE-2016-4490	Write Access Violation

Table 4.4: List of vulnerabilities in *binutils* tested and its type.

We trained three agents, each based on a different reward function, as described in section 3.4.3. We have used DDQN algorithm to train each agent, and the configuration of the network has been kept unchanged from the last section. Each agent trained for 48 hours. During testing, we ran each experiment 5 times with a timeout of 6 hours.

To our surprise, none of the agents were able to find an input that reached the target location within the stipulated time. None of the agents were able to learn an effective mutation strategy from the experiences. We have listed some of the possible reasons for this failure in chapter 6.

Chapter 5

Related Work

In this chapter, we discuss some of the popular techniques used for directed testing and how it differs from our approach. Later in the chapter, we analyze some machine learning based techniques that are used for program testing.

5.1 Symbolic Execution Based Approaches

Symbolic execution is the choice of technique for directed testing. The underlying idea in symbolic execution is to infer a logical formula representing the program execution. For directed testing, tools based on symbolic execution often leverage concrete executions. We discuss some of the influential works in this area.

1. **Directed Symbolic Execution** - This work [1] studied the problem of directed testing as a line reachability problem. Given a target line in the program, it tries to find an input which reaches that program point. It has proposed and evaluated three different strategies for directed symbolic execution. The first one is the $shortest - distance\ symbolic\ execution$ (SDSE). It uses the distance from the interprocedural control flow graph to guide the symbolic execution in a top-down approach. The second one is the $call - chain - backward\ symbolic\ execution$ (CCBSE). In this technique, the tool starts from the target line and move backward to reach the start of the program. This technique doesn't scale well, so a third technique, $mixed - strategy\ CCBSE$ (Mix-CCBSE) is introduced. Here, the CCBSE technique is used alternately with a separate forward strategy like KLEE[30]. The results show that the SDSE strategy works well in most cases, but it fails to reach the target location for all the test programs. Overall the Mix-CCBSE strategy was able to generate test cases reaching the target location in more number of programs.

2. **KATCH** - KATCH [2] attempts to test software patches using symbolic execution, aided with several heuristics. It starts by taking a set of inputs to the program and selects the one which is closest to the patch based on the interprocedural control flow graph. It uses symbolic execution and iteratively modifies the chosen input to reach the patch. Three heuristics, namely greedy exploration, informed path regeneration, and definition switching, are used to aid symbolic execution. These heuristics are based on the control and data dependencies of the program. Although KATCH increases patch coverage significantly, it fails to reach a considerable amount of target locations in the program.

3. **Driller** - Driller [31] is a hybrid vulnerability excavation tool which tries to achieve higher code coverage. It employs concolic execution and fuzzing in a complementary manner to find bugs hidden deep in the program. It starts with a random fuzzer to explore the paths in the program. Whenever the fuzzer gets stuck and is unable to find new paths, the concolic execution engine kicks in and generates a new input which can guide the fuzzer to the next "compartment". Combining concolic execution with fuzzing negates some of the limitations of any symbolic execution based tool. Despite some limitations, it was able to find more vulnerabilities in the program, compared to both concolic execution and random fuzzing.

Despite being the most popular technique, symbolic execution based tools have many limitations. Path explosion, inability to scale to large programs, and dependencies on underlying SMT theories, calls for better alternative solutions. Also, tools based on symbolic execution have to employ heavyweight program analysis which requires a deep understanding of the language semantics and underlying memory models. Our proposed approach, which is data driven, doesn't depend on such heavyweight program analysis. We apply lightweight instrumentation in the program to collect data about the flow of input data during program execution. This data is used by an RL agent to learn a better strategy and increase the efficiency of the fuzzer.

5.2 Taint Tracing Based Approaches

Taint analysis is another choice of technique for directed testing. The goal of tools based on taint analysis is to find bytes in the input to the program which influences the program execution. It works on the belief that not all bytes of the input to the program have an equal impact on the program execution path. For directed testing, taint analysis is often combined with fuzzing techniques. Taint analysis library tells the fuzzer which input bytes are useful and which are not. The fuzzer skips perturbing the bytes that do not affect the program

execution.

BuzzFuzz [32] uses dynamic taint tracing to find the significant bytes in the program input automatically. It records the input bytes influencing each computation in the program. BuzzFuzz inserts call to its taint tracing library at appropriate locations in the program to record this information. A random fuzzer uses this information to mutate only the influential bytes as provided by the BuzzFuzz library. Although BuzzFuzz was able to reach deeper in the program, it was unable to find bugs in the program. The performance overhead of the taint tracing library, a 20-30 times slowdown, neutralizes its efficiency to find bugs.

5.3 Heuristics Based Approaches

Here we discuss some of the recent works in the area of directed testing where fine-tuned heuristics were employed to existing techniques to achieve their goal.

1. **VUzzer** - VUzzer [33] is coverage based fuzzer and applies *application-aware* evolutionary fuzzing strategy. It combines various data flow and control flow features to aid the fuzzer. It also employs dynamic taint analysis to capture common characteristics of valid inputs, such as magic bytes, and error handling code. These heuristics helps the fuzzer to find bugs quicker than the baseline fuzzer. Although applying such lightweight techniques to aid the fuzzer results in increased performance, we still need to find the best-performing heuristics. Alternatively, a data-driven approach doesn't require such fine tuning as it can learn from past experiences.

2. **AFLGo** - AFLGo [12] is a directed greybox fuzzer based on the popular tool AFL [4]. AFLGo casts reachability as an optimization problem. The objective of the fuzzer is to decide the scheduling time of each seed input based on its seed distance. A seed distance for a program input is defined as a harmonic mean of the distances of each basic block from the target basic block. Here the distance is computed from the interprocedural control flow graph of the program. It employs *simulated annealing* as a *meta-heuristic* to solve the power scheduling of the seed inputs. AFLGo was able to reach the target location in the program most of the time, but it failed to do so in a consistent manner. Due to the random nature of fuzzing tools, it is tough to evaluate and compare these tools empirically. Also, tools like these require us to continually look out for better heuristics which can outperform the current one.

3. **Angora** - Angora [7] is coverage based fuzzer and doesn't explicitly perform directed testing. It tries to solve path constraints, as generated by any symbolic execution based

tool, without using symbolic execution. It uses techniques like context-sensitive branch coverage, and byte-level taint tracking to explore the program and generate path constraints. It adopts gradient descent algorithm, which is a prominent technique in many machine learning algorithms, to solve these path constraints, Also, to increase the efficiency of gradient descent algorithm, it tries to infer the type and shape of bytes in the program input. It was able to increase code coverage substantially and find more bugs in the various benchmarks, as compared to AFL [4].

4. **Hawkeye** - Hawkeye [13] is a directed grey-box fuzzer. It builds on top of techniques used in AFLGo [12] by adding several heuristics and additional program information. It strives to achieve a balance between limitations and benefits in static and dynamic analyses. To reach the target location quickly, it employs heuristics like power scheduling, adaptive mutation strategy, and seed prioritization. Although it was able to reach the target location quicker than ALFGo, it still suffers from the same limitations of AFLGo, i.e., to find the right mix of heuristics to employ for best performance. Our approach, on the other hand, is data-driven and doesn't require much external fine-tuning.

5.4 Machine Learning Based Approaches

In recent years, machine learning, and its application in various areas have garnered much attention. People have tried to apply machine learning techniques for program testing purposes also. We discuss some of the relevant works in this area.

1. **Learn&Fuzz** - Learn&Fuzz [14] is one of the earliest works which applies machine learning for input fuzzing. This work explores the possibility of learning the program-input grammar from sample inputs. Blind or random fuzzing suffer from the problem of generating useless inputs which almost always fail to pass the initial checks. The random fuzzer doesn't know about the grammar of the program input and often mutates parts of the input which leads to malformed input to the program. Learn&Fuzz also learns which parts of the input, if mutated, are more likely to generate useful test cases. Although there is a conflict between the learner and the fuzzer, as the learner wants to keep the structure valid and the fuzzer wants to break that structure to find bugs, it was still able to learn the structure of PDF inputs. This work shows that a data-driven approach is also useful for fuzzing purposes.

2. **Not all bytes are equal** - This work [34] explores the possibility of finding the optimal locations in the program input to mutate. It starts by gathering data generated by a

grey-box fuzzer like AFL [4]. This data is then used by a neural network architecture to learn a function that can predict these optimal locations. It works on the bit-level and generates a heat-map of the input bits. The fuzzer uses this heat-map to make informed choices in selecting the location of the next mutation. Architectures like LSTM (Long Short Term Memory), [35] and sequence-to-sequence with attention [36] were evaluated. Overall, the LSTM architecture was found to be most effective among all others in increasing code coverage over the baseline AFL across a variety of programs.

3. **NEUZZv1** - NEUZZv1 [15] seeks to learn a neural program which approximates the actual program under test. Based on the learned neural program, it tries to find a mutation strategy that maximizes code coverage. To learn this neural program, it learns the association between the control flow edges and the program input bytes. Convolution Neural Networks (CNN), a prominent architecture to learn associations, are used for this purpose. The result shows huge improvements both in terms of code coverage and finding new bugs in the program.

4. **NEUZZv2** - NEUZZv2 [37] improved upon the work of NEUZZv1. It moved to a simpler Neural Network architecture and introduced neural program smoothing. The idea is to learn a "smooth" version of the neural program that helps the optimization algorithms. It was able to find more bugs and increase code coverage in target programs as compared to the existing state of the art fuzzers.

5. **Deep Reinforcement Fuzzing** - This work [38] tries to adopt reinforcement learning for fuzzing purposes. It formalizes input fuzzing as a Markov decision process [39]. Based on past experiences, it learns an efficient mutation strategy to increase code coverage. It encodes a fixed length sub-string of the program input as a state for the agent, and the set of mutations are the set of possible actions for the agent. The agent receives either a reward or a penalty based on the coverage and/or execution time information. Results indicate that the agent was able to learn the task and improved upon the baseline random fuzzer.

6. **FuzzerGym** - FuzzerGym [40] also applied reinforcement learning to optimize mutation operators for efficient fuzzing. It encodes the whole program input in the form of a bit sequence as a state for the agent. For rewards, it uses a simple metric of line coverage, i.e., if the mutation were able to increase the line coverage, it would reward the agent; otherwise, it would penalize it. With 8 hours of training time, it was able to outperform the baseline fuzzer, which shows that it is possible to learn effective mutation strategies.

7. **Adaptive Grey-Box Fuzz Testing with Thompson Sampling** - This work [41] attempts to improve the mutation-selection strategy to increase the efficiency of the fuzzer. It casts the selection of a mutation operator as pulling an arm in a multi-armed bandit problem. It uses Thompson sampling [26] to adaptively learn a distribution over the mutation operators. The results show that there were significant improvements in the fuzzer performance.

All the works mentioned above are coverage oriented and not directed towards a particular location. It shows that machine learning can help to increase code coverage and find bugs. It also shows that data-driven approaches can be effective for a variety of learning tasks ranging from learning the program input grammar to finding efficient mutation strategies. Our work builds on top of these observations and tries to find an efficient mutation strategy for directed testing.

Chapter 6

Conclusion and Future Work

This thesis presents a data-driven approach for directed fuzz testing. The key insight of the thesis is to leverage the data generated by fuzz testing tools to create a better mutation strategy. We use reinforcement learning to train an agent for this purpose. Our experiments show that for simple programs, the agent can learn a better mutation strategy based on the given program input. However, for complex programs, it is not able to perform better than a random strategy and often fails to reach the target location.

We detail some of the reasons that we suspect contributed to the failure of our approach. First, any agent trying to learn from experiences need a lot of positive rewards for training. Most of the mutated program inputs fail basic sanity checks and result in fewer positive rewards. For example, in *libxml2*, the mutations resulting in positive rewards constitute only 2-5% of all mutations. The agent was not able to gain any information from the very few positive rewards.

Second, the reward metrics did not convey enough information about the environment to the agent. RL agents are sensitive to reward functions and should be selected carefully. We had tested our approach with three different reward functions, and it is not an exhaustive list by any means. In some experiments, we see that one of the agents is performing better than others. We intend to explore it further with a variety of reward functions that provide more information about the environment.

Third, the state was not well represented. For our experiments, we selected the sequence of bytes of the program input to represent the state. Although the agents are not very sensitive to the state representation like reward functions, more informative representation of state could have helped.

Fourth, the agent had very less control over the decision making. In our approach, the agent only had control over the mutation to be applied. However, the fuzzer was free to

choose the location of the mutation. So even if our agent responded with an appropriate mutation, applying it at a different position could result in negative or nil rewards.

Although our proposed approach was not able to perform as good as the state-of-the-art fuzzers, we learned some valuable lessons. We hope that this thesis acts as a stepping stone for anyone seeking to use a data-driven approach for directed fuzzing.

Future Work

We have identified some of the shortcomings in our approach and intend to follow up on it. Some of the areas that can be looked upon are:

- **Reward Functions**: As the agents are very sensitive to the reward function, one can explore more reward functions that provide more information about the state of the environment.
- **State Representation**: A simple byte sequence can be combined with more information about the program. This information can be obtained from the program execution, e.g., function call sequence, complete trace, call stack.
- **Combining States**: Presently, every program input is a different state for the agent. However, many program inputs take the same execution path and can be combined to form a single state.

References

[1] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011. 1, 45

[2] Paul Dan Marinescu and Cristian Cadar. KATCH: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 235, 2013. ISBN 9781450322379. doi: 10.1145/2491411.2491438. URL http://dx.doi.org/10.1145/2491411.2491438.http://dl.acm.org/citation.cfm?doid=2491411.2491438. 1, 2, 46

[3] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 1

[4] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2018. [Online; accessed Sunday 29th December, 2019]. 2, 3, 7, 8, 47, 48, 49

[5] LLVM. libFuzzer. https://llvm.org/docs/LibFuzzer.html, 2018. [Online; accessed Sunday 29th December, 2019]. 2

[6] Peach Tech. Peach Fuzzer. http://www.peach.tech/products/peach-fuzzer/, 2018. [Online; accessed Sunday 29th December, 2019]. 2

[7] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. URL https://arxiv.org/pdf/1803.01307.pdf. 2, 47

[8] Google. Syzkaller. https://github.com/google/syzkaller, 2018. [Online; accessed Sunday 29th December, 2019]. 2

[9] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017. 2

[10] Michal Zalewski. American Fuzzy Lop : trophies. http://lcamtuf.coredump.cx/afl/#bugs, 2018. [Online; accessed Sunday 29th December, 2019]. 2, 8

[11] LLVM. libFuzzer : Trophies. https://llvm.org/docs/LibFuzzer.html#trophies, 2018. [Online; accessed Sunday 29th December, 2019]. 2

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, pages 2329–2344, 2017. ISSN 15437221. doi: 10.1145/3133956.3134020. URL http://dl.acm.org/citation.cfm?doid=3133956.3134020. 3, 20, 47, 48

[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108. ACM, 2018. 3, 48

[14] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017. 3, 48

[15] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018. 3, 49

[16] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017. 3

[17] TechCrunch. Google's AlphaGo AI wins three-match series against the world's best Go player. https://techcrunch.com/2017/05/24/alphago-beats-planets-best-human-go-player-ke-jie/, 2018. [Online; accessed Sunday 29th December, 2019]. 4, 11

[18] The Guardian. AlphaZero AI beats champion chess program after teaching itself in four hours . https://www.theguardian.com/technology/2017/dec/07/

alphazero-google-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-
2018. [Online; accessed Sunday 29th December, 2019]. 4, 11

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. URL https://www.cs.toronto.edu/{~}vmnih/docs/dqn.pdf. 4, 5, 11, 15, 16

[20] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 3389–3396. IEEE, 2017. 4, 11

[21] MA Wiering. Multi-agent reinforcement learning for traffic light control. In *Machine Learning: Proceedings of the Seventeenth International Conference (ICML'2000)*, pages 1151–1158, 2000. 4, 11

[22] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015. 5, 28

[23] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016. 5

[24] Caca Labs. zzuf - Caca Labs. https://github.com/samhocevar/zzuf, 2016. [Online; accessed Sunday 29th December, 2019]. 7

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005. doi: 10.1145/1065010.1065036. URL http://doi.acm.org/10.1145/1065010.1065036. 7

[26] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on Learning Theory*, pages 39–1, 2012. 18, 50

[27] Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 2019. [Online; accessed Sunday 29th December, 2019]. 26

[28] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016. 28

[29] Arthur Juliani. DeepRL-Agents. https://github.com/awjuliani/DeepRL-Agents, 2017. [Online; accessed Sunday 29th December, 2019]. 28, 31

[30] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. 45

[31] 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 *NDSS*, volume 16, pages 1–16, 2016. 46

[32] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 474–484. IEEE Computer Society, 2009. 47

[33] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017. 47

[34] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017. 48

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 49

[36] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. 49

[37] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *NEUZZ: Efficient Fuzzing with Neural Program Smoothing*, page 0. IEEE, 2018. 49

[38] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 116–122. IEEE, 2018. 49

[39] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. 2011. 49

[40] William Drozd and Michael D Wagner. Fuzzergym: A competitive framework for fuzzing and learning. *arXiv preprint arXiv:1807.07490*, 2018. 49

[41] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 37–47. ACM, 2018. 50