A STUDY OF GRAMMAR-BASED FUZZING APPROACHES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ziwei Wu

June 2022

COMMITTEE MEMBERSHIP

TITLE: A Study of Grammar-Based Fuzzing Approaches

AUTHOR: Ziwei Wu

DATE SUBMITTED: June 2022

COMMITTEE CHAIR: Phoenix Fang, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Stephen Beard, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Rodrigo Canaan, Ph.D.
Professor of Computer Science

ABSTRACT

A Study of Grammar-Based Fuzzing Approaches

Ziwei Wu

Fuzzing is the process of finding security vulnerabilities in code by creating inputs that will activate the exploits. Grammar-based fuzzing uses a grammar, which represents the syntax of all inputs a target program will accept, allowing the fuzzer to create well-formed complex inputs. This thesis conducts an in-depth study on two blackbox grammar-based fuzzing methods, GLADE and Learn&Fuzz, on their performance and usability to the average user. The blackbox fuzzer Radamsa was also used to compare fuzzing effectiveness. From our results in fuzzing PDF objects, GLADE beats both Radamsa and Learn&Fuzz in terms of coverage and pass rate. XML inputs were also tested, but the results only show that XML is a relatively simple input as the coverage results were mostly the same. For the XML pass rate, GLADE beats all of them except for the SampleSpace generation method of Learn&Fuzz. In addition, this thesis discusses interesting problems that occur when using machine learning for fuzzing. With experience from the study, this thesis proposes an improvement to GLADE's user-friendliness through the use of a configuration file. This thesis also proposes a theoretical improvement to machine learning fuzzing through supplementary examples created by GLADE.

ACKNOWLEDGMENTS

Thanks to:

- Andrew Guenther, for uploading this template
- My advisor Dr. Fang, for her advice and time.
- My committee members Dr. Beard and Dr. Canaan, for their time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

In a world where many people are connected through the internet, cybersecurity is an important issue. One of the important aspects of cybersecurity is to make sure that the software we use do not contain any vulnerabilities. Software vulnerabilities can cause large amounts of monetary damage. The CSI security survey reported the following losses due to vulnerabilities: $378 million in 2001, $456 million in 2002, and $202 million in 2003 from 196, 223, and 252 respondents, respectively [20]. On a global scale, the costs of cyber attacks have become more costly. The Center for Strategic and International Studies, a nonprofit bipartisan policy research group, estimated that the cost of global cybercrime was $300 billion in 2013 and has increased to $945 billion in 2020 [31].

Software exploits can also cause non-monetary damages, such as loss of privacy or having personal data (e.g., photos) stolen. Historically, government-lead cyberattacks have been used to steal information, such as intellectual property and financial plans [37].

Among the vulnerabilities, 36% are due to configuration errors and design problems [20]. The remaining vulnerabilities are due to programming errors [20]. In recent years, there have been a further increase in the number of software vulnerabilities. According to the Common Vulnerabilities and Exposures (CVE) group, there were 6,457 published CVE records in 2016, which exploded to 14,645 CVE records in 2017 [2]. The number of CVE records has steadily increased to 20,161 in 2021 [2].

To prevent vulnerabilities, software developers can either create updates to their software as vulnerabilities are known or try to prevent them during software development. Patching software can be expensive, costing up to $100,000 [20]. Creating software updates can also be risky; if it has not been tested enough, then it may cause strange behaviors with the software [20]. On the other hand, delaying the update will allow attackers to use the vulnerability for a longer period of time [20].

The further software is into the development phases, the more expensive fixing a bug is. Some sample numbers from Google are that it takes $5 to fix a bug during unit testing and $5,000 to do so during system testing [13]. The IBM System Science Institute reported that bugs were 15 times more costly to fix during testing than during the design phase [11]. In addition, they found that patching a bug in the maintenance phase was 100 times more costly than having fixed it in the design phase. As such, preventing vulnerabilities in the early stages of software development is the best option.

Software analysis to detect vulnerabilities can be either static or dynamic. Static code analysis uses tools that inspect the code without executing the code [20, 33]. The methods used can range from rule matching, control-flow analysis, and many more [33]. A survey found that users at Siemens had positive experiences with the static analysis tools they used as the tools could point to where a violation occurred and give detailed reports [39]. The violations also had straightforward fixes [39]. One of the main drawbacks of static analysis tools used to detect vulnerabilities is the high number of false positives in large codebases, requiring the user(s) to manually check through the results [20, 33, 39]. Users of static analysis tools consider running the tools and fixing violations to be bigger constraints [39]. Having to perform manual checking for false positives would only add to the constraint.

Dynamic analysis involves testing the code during execution. Inputs are created to test the code by trying to go through, ideally, all execution paths [20]. Fuzzing is a technique to automatically generate test inputs, eliminating the need for the user to create test inputs. Unlike static analysis, fuzzing has no false positives [33]. Some drawbacks of fuzzing is that it can have a high false negative and high randomness. There is no way to be certain that a set of test inputs contain all possible program executions [16]. In addition, if a program takes a long time to run (e.g., hours), then trying to test all possible paths becomes expensive quickly.

In their definition of dynamic analysis, Heffley and Meunier did not mention penetration testing [20], but it is also technically a way to test a system during execution. Penetration testing involves a plan to enact how an attacker would penetrate the system and testers to carry out the attack [33]. Unlike fuzzing, the results of penetration testing depends on the testers' abilities and is not automatic [33].

1.1 Motivation

As mentioned above, fuzzing is an automatic way of generating inputs for a target program to test said program. The goal is to execute as many execution paths in the target software as possible, if not all execution paths. Among the popular ways to test software, fuzzing is automatic and does not require deep knowledge of the target program.

There are three main types of fuzzing techniques: blackbox random fuzzing, whitebox constraint-based fuzzing, and grammar based fuzzing [15]. Whereas whitebox and blackbox fuzzing have been fully automated, this is not true for grammar-based fuzzing. Grammar-based fuzzing requires an input grammar; an input grammar is

used to specify the format of the inputs [15]. The grammar must be manually built, which is labor-intensive and requires in-depth knowledge of the program.

Despite the fact that it is not fully automated, grammar-based fuzzing is the most effective technique for fuzzing programs that take highly structured inputs, such web browsers [15]. In addition, the most important use of grammar-based fuzzing has been testing compilers [46]. CSmith, a state-of-the-art tool for generating C programs to test C compilers, creates random C programs by using a grammar implemented via handcoded C++ classes [44]. Over three years of using CSmith to find bugs, Yang et al. have found more than 325 bugs in mainstream C compilers [44]. And on CSmith's website, the authors claimed to have found more than 400 previously unknown compiler bugs [43].

1.2 Challenges

One might assume that popular programming languages and formats would already have their grammars documented and ready for grammar-based fuzzing. Such documentation may be poorly written. For example, the documentation for the input syntax Flex and Bison are limited to informal documentation [7]. And when documentation is available, they are not written in a machine-friendly way. For example, Godefroid et al. report that the specification of the PDF format is a 1,300 page long PDF file [15].

CSmith, the grammar-based fuzzer for C compilers, was developed over several years and consists of about 41,000 lines of handwritten C++ code [10]. Because the logic behind the generation of the inputs is tightly coupled with the target program, each feature of the grammar has to be engineered specifically for the target program [10]. Con-

verting CSmith to CLSmith for fuzzing OpenCL compilers took about nine months and eight thousand additional lines of code [10].

As grammar-based fuzzing is held back by needing the manual creation of an input grammar, current work has proposed methods to automate the grammar creation process. However, there have not been studies about the advantages and disadvantages of the current state-of-the-art methods to automatically infer and learn a grammar.

1.3 Contribution

Therefore, in this thesis, the above challenges will be studied and discussed. The contributions of this thesis are summarized as follows:

1. A comprehensive study of automating grammar-based fuzzing methods is carried out.
2. A comparison of current grammar-based fuzzing methods are presented in terms of their advantages and disadvantages.
3. An improvement to current machine learning approaches by using grammar inference algorithms is proposed.
4. An improvement to the user-friendliness to GLADE is presented.

Chapter 2

BACKGROUND

In this chapter, I will give an overview of the topics of fuzzing, grammar, types of grammar, previous grammar inference algorithms, and an explanation of different machine learning methods. The goal of this chapter is not to give a comprehensive overview of all the topics, but give the readers enough knowledge to understand the principals behind fuzzing, grammars, and what machine learning models have been used in fuzzing.

2.1 Fuzzing

Fuzzing is a process of finding security vulnerabilities in code that parses inputs [15]. The target programs are tested by using modified, or *fuzzed*, inputs [15]. The inputs are created automatically [26] and the inputs are either random or semi-valid [33]. Random inputs are inputs that are randomly generated and semi-valid inputs are inputs that are valid enough to pass input checks but are still invalid enough to potentially cause problems [33]. Valid inputs are inputs that the target program accepts with no problems.

There are three main types of fuzzing techniques used today: blackbox, whitebox, and grammar-based [15, 26]. Blackbox fuzzing uses valid inputs and modifies them randomly, usually at the byte level [26]. It does not require expert knowledge of the code to use. The inputs generated by blackbox fuzzers may not be valid, resulting in lower code coverage.

Whitebox fuzzers use the structure of the code to form inputs, usually through symbolic execution to capture path constraints and using a constraint solver to generate inputs [26]. Because these fuzzers have full access to the code, they can guide the generated inputs to unexplored execution paths, resulting in higher coverage. The inputs generated by whitebox fuzzing are mostly valid [26].

This thesis focuses on grammar-based fuzzing, which will be explained next.

2.1.1 Grammar-based Fuzzing

Grammar-based fuzzers target programs that take in highly structured inputs [26]. An example of such a program would be a compiler for a programming language. A grammar-based fuzzer uses an input grammar to generate inputs; an input grammar specifies all valid inputs to the target program. As such, all generated inputs are valid to the program.

The main problem of grammar-based fuzzers that current research is trying to solve is its automation. An input grammar needs to be specified and trying to create a grammar manually takes a lot of work and requires in-depth knowledge of the target program. This thesis goes over the current work to automate the grammar generation process and compares them.

2.2 Machine Learning in Fuzzing

Wang et al. reviewed the research progress of using machine learning (ML) techniques for fuzzing in the recent years [40]. While there are many techniques for different use cases, the use case that will be focused on is what Wang et al. labeled as "testcase generation", specifically, "generation-based testcase generation". This use case fo-

cuses on using machine learning techniques to automatically learn the syntax from a collection of samples that conform to a grammar [40]. While there are many papers that use ML for fuzzing in this way, this thesis will only focus on a few prominent ones.

Learn&Fuzz, proposed by Godefroid et al., is the first attempt to learn the grammar of PDF files using the long short-term memory (LSTM) architecture of recurrent neural networks (RNN) [15]. To improve upon Learn&Fuzz, Jitsunari et al. proposed the use of coverage metrics to iteratively improve the model [26]. DeepSmith, proposed by Cummins et al., uses a LSTM to learn from open source OpenCL programs to fuzz OpenCL compilers [10]. In essence, it is nearly identical to Learn&Fuzz in the learning portion. The main difference is that voting heuristics were used to find anomalies in the different compilers. In addition, DeepSmith used a different encoder. DeepSmith is included because Jitsunari et al. compared it to their proposed solution.

Other than the above mentioned works, there are also many papers that use RNNs/LSTMs to fuzz by learning from data. The ML models have also been used to fuzz web browsers [38], network protocols [12], C compilers [34], and JavaScript engines [30].

Later in this chapter, I will give a brief overview of how RNNs and LSTMs work. After that is an explanation of reinforcement learning, as that is what Wu et al. used to improve upon a grammar inference algorithm [42].

2.3 Grammar

When talking about program inputs, a grammar is used to describe the inputs' structure. Figure 2.1 shows an example of a grammar for an integer.

<start> ::= <number>
<number> ::= <integer> | + <integer> | − <integer>
<integer> ::= <digit> | <digit><integer>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.1: Integer grammar example from [46].

The example grammar consists of a *start symbol* and a set of *expansion rules* [46]. Expansion rules are also referred to as *production rules*. Figure 2.1 has the expansion rules denoted by <A>::=<B>. Expansion rules signify that the symbol on the left, <A>, can be replaced by the symbol on the right, <B> [46]. A symbol can be replaced by multiple symbols together, such as <A> ::= <B><C> meaning that <A> can be replaced by <B> followed by <C>. A symbol can also have alternate expansions. For example, <A>::=<B> | <C> means that <A> can be expanded into either <B> or <C>, with the "|" operator being used to represent "or" in this instance.

In the example grammar, all symbols are denoted by being inside of angle brackets, such as <start> or <number>. Anything not inside the angle brackets are literal values.

The full explanation of the grammar in Figure 2.1 is as follows. The start symbol expands into the symbol representing a number and the number symbol has three different expansion rules: an integer symbol, an integer symbol preceded by a plus sign, and an integer symbol preceded by a minus sign. An integer symbol consists of two expansion rules: a digit or a digit followed by an integer. By having itself in one of the rules, the integer symbol can be represented as having 1 or more digits through recursion. And finally, a digit is a number from 0 to 9.

By beginning at the start symbol and going through all the expansion rules, you can create valid integers as explained by the grammar. For example, "-32", "007", and "+127" are all values that can be created by going through the grammar rules.

As the inputs for a program become more complicated, so does the grammar. And trying to write the grammar for something as complicated as a programming language from scratch would be unfeasible without deep knowledge of said programming language.

2.4 Types of Grammar

This section will explain the two types of grammar most commonly used, context-free grammar and probabilistic context-free grammar.

2.4.1 Context-Free Grammar

The context-free grammar (CFG) is the most widely used formal system for modeling the structure of English and other natural languages [28]. It can also be used to represent the structure of other things, such as program inputs.

The grammar in Figure 2.1 is an example of a context-free grammar. The formal definition is as follows. A context-free grammar G is defined by the four parameters: N, Σ, R, and S [28].

N is a set of non-terminals symbols. A non-terminal symbol in Figure 2.1 is any of the symbols inside the angle brackets, such as <start>. They are non-terminal symbols because they can be further expanded through the grammar rules.

Σ is a set of terminal symbols that are disjoint from N. In the case of Figure 2.1, terminal symbols are anything not inside the angle brackets. For example, the plus sign, the minus sign, and the numbers that a digit can be. The terminal symbols can not be expanded further and are taken as literal values.

R is a set of expansion rules in the form A $\rightarrow \beta$, where A is a non-terminal symbol and β is a string of symbols from the set (N $\cup$ Σ)*. The "*" symbol represents 0 or more repetitions. It is possible that a non-terminal symbol expands into nothing, usually represented as ε. In Figure 2.1, the arrow symbol is replaced with "::=".

S is the start symbol and is a member of N.

2.4.2 Probabilistic Context-Free Grammar

A Probabilistic Context-Free Grammar (PCFG) is a CFG with a probability distribution [42]. A PCFG G is defined by the five parameters: N, Σ, R, S, and P.

All the parameters represent the same thing as the formal definition of a CFG. The only thing that is added is P, which is the set of probability distributions over the production rules. For a non-terminal A, its k number of production rules are defined as r_1^A, ..., r_k^A [42]. The probability of an expansion rule r_i^A being chosen is $P_A(i)$, where $P_A(i) \in [0, 1]$ and i = $\{1, ..., k\}$ [42]. In addition, $\sum_{i=1}^{k} P_A(i) = 1$ must be satisfied [42].

An example of a PCFG is in Figure 2.2. Rather than using the "|" operator, the different expansion rules are on their own lines. The only difference is that there is now a probability associated with each rule, represented by the number in parentheses after each rule. The probabilities were chosen arbitrarily and only used to demonstrate what a PCFG would look like.

<start> ::= <number> (1.0)
<number> ::= <integer> (0.5)
<number> ::= + <integer> (0.1)
<number> ::= − <integer> (0.4)
<integer> ::= <digit> (0.1)
<integer> ::= <digit><integer> (0.9)
<digit> ::= 0 (0.1)
<digit> ::= 1 (0.1)
<digit> ::= 2 (0.1)
<digit> ::= 3 (0.1)
<digit> ::= 4 (0.1)
<digit> ::= 5 (0.1)
<digit> ::= 6 (0.1)
<digit> ::= 7 (0.1)
<digit> ::= 8 (0.1)
<digit> ::= 9 (0.1)

Figure 2.2: Integer grammar example from [46] modified into a PCFG.

2.5 Grammar Inference Algorithms

There are multiple grammar inference algorithms, with Bastani et al. pointing to the two most studied algorithms: L-star and RPNI [7].

2.5.1 L-star

Angluin introduced the learning algorithm L^* to identify an unknown regular set (e.g., a grammar) using membership queries and counterexamples [4]. The algorithm uses a source of examples called the *Teacher* and the learning algorithm is called the *Learner*. The *Learner* learns by asking the *Teacher* two types of questions and receiving the answers.

1. Is this string a member of the unknown grammar (i.e., can it be generated using the grammar)? Here, the *Teacher* will reply with either a *yes* or *no*.

2. Is this inferred grammar equivalent to the unknown regular set? If it is, then the *Teacher* will reply with a *yes*. Otherwise, the *Teacher* will give a counterexample.

The *Learner* begins by asking the *Teacher* membership queries until a condition is met. Afterwards, the *Learner* will construct an inferred grammar and ask the *Teacher* if it is correct. If no counterexamples are given, then the inferred grammar is returned. Otherwise, the *Learner* starts again by asking the *Teacher* membership queries.

In addition, Angluin writes that the second part of the *Teacher* (searching for a counterexample) is too restrictive and can be replaced with a random sampling oracle. The alphabet that is used to create all strings that is accepted by the unknown grammar is A and the unknown grammar is a subset of A^*. The random sampling oracle samples a string from the set A^* and returns (x, d), where x is an element and d is whether or not the element is a member of the unknown grammar. If d is *yes* but x is not a member of the inferred grammar or vice versa, then it is a counterexample. For Bastani et al., the inferred grammar was accepted if there were no counterexamples after 50 samples [7].

2.5.2 Regular Positive Negative Inference

The full explanation of RPNI is beyond the scope of this thesis. In short, RPNI learns using positive examples and negative examples, with positive examples being examples that are members of the unknown grammar and negative examples being examples that aren't members [7, 35].

2.5.3 Problems with the Above Grammar Inference Algorithms

As stated by Bastani et al., the grammar inference algorithms L-star and RPNI consistently overgeneralize and undergeneralize [7]. Suppose that Σ represents an input alphabet (e.g., ascii characters) and a target program takes in Σ^* as input. L_* denotes all inputs that the target program accepts, where $L_* \subseteq \Sigma^*$.

Overgeneralization is when the grammar that is inferred produces inputs outside of what is accepted by the target program (e.g., the inferred grammar is Σ^*) [7]. Undergeneralization is the opposite problem, where the grammar inferred is unable to create all inputs accepted by the program (e.g., the inferred grammar is $\emptyset$) [7].

2.6 Recurrent Neural Networks

Recurrent neural networks (RNNs) work in the following manner [15]. A RNN is a neural network that operates on an variable length input $x = (x_1, x_2, ..., x_T)$ and each neuron/node has a hidden state h. At a time stamp t, the hidden state h_t is updated using the previous hidden state h_{t-1} and the input x_t. The output y_t is calculated using the new hidden state as shown in Equation 2.1 [15]. In Equation 2.1, f is a non-linear activation function and ϕ is a function that calculates the output probability distribution.

$$\begin{aligned} h_t &= f(h_{t-1}, x_t) \\ y_t &= \phi(h_t) \end{aligned} \tag{2.1}$$

Equation 2.1: Equations for a recurrent neural network [15].

After the hidden state has been updated to h_t, the new hidden state can be used to generate a probability distribution for the next elements of the sequence [9]. That is,

the probabilities $p(x_t|x_1, x_2, ..., x_{t-1})$ are outputted, where $(x_1, x_2, ...x_{t-1})$ were previous elements of the sequence [9].

2.7 Long Short-Term Memory

One problem with standard RNNs is that they have "amnesia", where information about the past inputs is lost [19]. A generative RNN works by outputting a probability distribution, choosing an element from the distribution, and using the chosen element as input. If a generative model's predictions is only based on the few past inputs and those past inputs were created by the RNN, then it will not be able to recover from its past mistakes [19].

Long Short-Term Memory (LSTM) is a solution to the this problem as they are designed to be better at storing and accessing information [19]. LSTM was originally introduced by Hochreiter and Schmidhuber in 1997 [22]. Since then, a number of modifications have been made. The formulation of the LSTM used by Graves (2013) is detailed in Equation 2.2 [19]. In Equation 2.2, o, c, f, and i are the output gate, memory cell, forget gate, and input gate activation vectors respectively [19]. The subscripts for the weights W are labeled with the activation vectors in mind. For example, W_{xo} is the matrix of weights for input x_t when calculating for o_t and etc. The σ symbol represents the logistic sigmoid function. The b at the end of each equation is the intercept for that gate.

$$\begin{aligned} h_t &= o_t tanh(c_t) \\ o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \\ c_t &= f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\ f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \\ i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \end{aligned} \tag{2.2}$$

Equation 2.2: Equations for a LSTM [19].

In short, the LSTM architecture can be thought of as a RNN with memory cells to store long range dependencies. The *input*, $forget$, and *output* gates are trained to learn what information to store in the memory cell, how long to store that information, and when to read it out, respectively [14].

2.8 Reinforcement Learning

Reinforcement learning (RL) is essentially learning through interactions [5]. An autonomous agent is in its environment at state s_t at time step t. The state of the environment is comprised of all necessary information about the environment the agent needs to choose the best action.The agent picks an action a_t to perform, changing the environment and agent to the new state s_{t+1}. After the transition to the new state, the environment gives the agent a scalar reward r_{t+1}. The goal of the agent is to learn a policy π that maximizes the expected reward, where a policy returns an action to perform given the current state of the environment.

The state having all necessary information is only true where the problem is simple enough to be formulated as a Markov Decision Process (MDP). However, this is unrealistic [5]. A generalization of the MDP is the Partially Observable MDP (POMDP), where the agent does not have all information of the state.

Chapter 3

RELATED WORK

This chapter will look into the current state-out-the-art methods of automating grammar-based fuzzing.

3.1 GLADE

Bastani et al. proposed an algorithm called GLADE to infer input grammars in the blackbox setting [7].

3.1.1 Current Solutions and Limitations

As mentioned previously, grammar-based fuzzing requires the use of an input grammar, which has to be manually created. In addition, input grammars can be hard to find and the grammars that are documented are not machine friendly.

There are two ways to synthesize a grammar: whitebox and blackbox. Whitebox requires the target program's code to be available for analysis, which may be unviable when only the executable parts of a program are available. As such, the authors went for the blackbox approach, where only the ability to execute the target program is required.

Language inference algorithms, the most studied being L-Star and RPNI, were found to be unsuitable by the authors for grammar inference. For example, the two algorithms were unable to learn simple input languages such as XML.

Current language inference algorithms make assumptions that do not hold true for learning program input grammars. For example, they rely on an "oracle" to provide negative examples to remove the overgeneralized parts of the grammar and positive examples that have contain all interesting behaviors. This causes the synthesized grammars to either be overgeneralized or undergeneralized.

3.1.2 Proposed Solution

The authors propose an algorithm, GLADE, for synthesizing program inputs which addresses the drawbacks of the current inference algorithms.

Given a small set of seed inputs and blackbox access to the target program for *membership queries*, the proposed solution will create a grammar for the inputs of the target program.

It addresses the shortcomings of existing algorithms by:

1. Considering more generalizations
2. Generates negative examples during runtime

These solutions prevent the problems of omitted positive examples and omitted negative examples, respectively.

3.1.3 Explanation of Algorithm

GLADE works by applying generalization operators to the seed inputs. The generalization operators used are *repetition* and *alternation*. Repetition means that a substring is changed to have 0 or more occurrences. Alternation changes a substring

to alternate between multiple choices. The generalizations are applied to substrings as the input string is decomposed into $\alpha = \alpha_1\alpha_2...\alpha_n$.

For example, the XML string <a>hello<\a> can have the repetition operator applied to it. The candidate language would be <a> $[\text{hello}]_{rep}$ <\a>. Examples of generated inputs from this candidate language are: "<a><\a>", "<a>hello<\a>", and "<a>hellohello<\a>".

The alternation operator can be then applied to <a> $[\text{hello}]_{rep}$ <\a>, creating the candidate language <a> $[(\text{he} + \text{llo})]_{rep}$ <\a>, where "(he + llo)" represents the choice of choosing either "he" or "llo". Example outputs are "<a>he<\a>", "<a>hehe<\a>", and "<a>llo<\a>".

For each i-th generalization step, GLADE wants to find a candidate language (inferred grammar) that improves recall without sacrificing precision. That is, $\hat{L}_{i+1} \backslash \hat{L}_i \in L_*$, where $\hat{L}_i$ is the candidate language at generalization step i and L_* is the unknown grammar. $\hat{L}_{i+1} \backslash \hat{L}_i$ is the set of elements that are in $\hat{L}_{i+1}$ but not in $\hat{L}_i$.

This is to make sure that overgeneralization does not happen. When a generalization is done, the candidate language generates input strings. Those input strings are checked whether or not they are accepted by the target program. If one input is rejected, then the candidate language is rejected. However, since there may be an infinite amount of producible strings, only a finite amount is checked.

Multiple candidate languages are created and tested at each step. The best one is chosen and further generalizations are applied. If GLADE is unable to perform another generalization step, then the current candidate language is returned. The candidate language is returned as a regular expression, which is then converted into a context-free grammar (CFG). The CFG has character generalizations applied to it, which allows terminal symbols to be replaced by other terminal symbols.

3.1.4 Input Generation

Bastani et al. use two methods to generate inputs from the inferred grammar: sampling and fuzzing. The authors describe the conversion of the CFG into a PCFG, where the expansion rules of each nonterminal have a uniform distribution. In other words, if a nonterminal has k expansion rules, then each rule has the probability of $\frac{1}{k}$. For each nonterminal, their expansion rules are chosen via sampling, or chosen based on their probabilities. From the start of the PCFG, the expansion rules are sampled until all have been expended into terminals.

For fuzzing, Bastani et al. uses the following method. First, the fuzzing method takes in two variables: the CFG and sample inputs. The sample inputs are the same ones GLADE used to infer a grammar. A random input, α, is chosen and a parse tree for it is created according to CFG. There are n modifications made to α, where n is a random number from 0 to 50. A single modification consists of the following:

- A node N is randomly chosen from the parse tree of α.
- α is decomposed into $\alpha_1\alpha_2\alpha_3$, where α_2 is the substring parsed by the subtree with node N as the root.
- The node N is represented as a nonterminal, A. A is sampled with the expansion rules of the CFG, creating the substring α'. α' replaces α_2 and $\alpha_1\alpha'\alpha_3$, is returned.

3.2 REINAM

Wu et al. propose REINAM, an algorithm to infer input grammars through the use of reinforcement learning [42].

3.2.1 Current Solutions and Limitations

Wu et al. point out challenges of current grammar inference algorithms:

1. Unanalyzable code: Some algorithms use static code analysis or info from instrumentation. These approaches cannot handle code that cannot be instrumented, such as web services, or code that is too difficult for static analysis to handle.
2. Low variety and quality of seed inputs: There are existing techniques that infer a grammar based on seed inputs. However, the quality of the grammar is reliant of the variety and quality of the seed inputs.
3. Lack of speed inputs: Machine learning techniques require a large amount of data. More often than not, there are not enough valid examples to learn from.

Another big challenge of the then current algorithms (e.g., GLADE) is the fact that candidate grammars are discarded if they produce an output that is rejected by the target program. While this prevents the grammar from overgeneralizing, it also prevents the potential for higher code coverage.

3.2.2 Proposed Solution

The authors propose REINAM, a framework that uses reinforcement learning to synthesize an input grammar while improving the performance of current state-of-the-art algorithms.

REINAM improves upon current state-of-the-art algorithms (the authors mostly mention GLADE) by doing three things:

1. Using a PCFG instead of a CFG.

2. Keeping candidate grammars even if they produce invalid inputs and using reinforcement learning to adjust the PCFG probabilities.

3. Using automatic test generation algorithms to create seed inputs.

3.2.3 Explanation of Algorithm

REINAM has two parts in how it operates and works as follows.

The first part of REINAM uses an automatic test generator to create high quality seed inputs to be used by an existing grammar inference algorithm. In the paper, Wu et al. uses Pex, a whitebox test generator for .NET programs, and has GLADE infer a grammar with the generated inputs.

The second part of REINAM improves upon the inferred grammar through the use of reinforcement learning. The grammar inferred by GLADE is first converted from a CFG into a PCFG. Each of the k expansion rules for a non-terminal A is given the probability $\frac{1}{k}$. After the conversion, generalization operators (the same ones used by GLADE) are used to add new rules into the existing grammar. When a new expansion rule is added for a non-terminal A, the new rule is given the probability of $\frac{1}{k}$, where k is the number of production rules. The probability of the other rules are reduced proportionally.

As a new rule is added by a generalization operator, the PCFG is used to create new inputs. As the i-th expansion rule r_i^A, for non-terminal A, is used to create inputs, the probability of how many created inputs are accepted by the target program is recorded as a reward, $reward(r_i^A)$. The reward is used to adjust the probabilities of the PCFG as shown in Equation 3.1. In Equation 3.1, $\theta^{t+1}(r_i^A)$ is the new probability for rule

r_i^A calculated using the current probability $\theta^t(r_i^A)$, $reward(r_i^A)$, and the learning rate η. The probabilities are then normalized such that $\sum_{i=1}^{k} \theta^t(r_i^A) = 1$.

The above step for a new rule is done until each of the probabilities do not change by the threshold $\frac{1}{10*k}$.

$$reward(r_i^A) = \frac{\text{\# of accepted inputs created by rule } r_i^A}{\text{\# of inputs created by rule } r_i^A}$$
$$\theta^{t+1}(r_i^A) = \theta^t(r_i^A) + (\eta * \frac{reward(r_i^A)}{\theta^t(r_i^A)}) \tag{3.1}$$

Equation 3.1: Equations for REINAM [42].

3.2.4 Input Generation

Wu et al. uses the same input generation method as GLADE, sampling. However, the sampling method does not make use of the PCFG probabilities. The authors explain that the probabilities are discarded because they do not represent the true distribution of inputs. Rather, the probability of an expansion rule represents how likely an input created using said expansion rule would be accepted by the target program. Because of this, the expansion rules of a each nonterminal have a uniform distribution instead. And as for fuzzing, Wu et al. also used the uniform sampling method.

3.3 Learn&Fuzz

Godefroid et al. proposes the use of machine learning techniques to automatically generate a grammar for grammar-based fuzzing using sample inputs [15].

3.3.1 Current Solutions and Limitations

The problem statement is similar to GLADE's. Grammar-based fuzzing is not automatic and require an input grammar to be made by hand, which takes a lot of time and may be prone to errors.

In addition, current algorithms for learning a grammar are not suitable for some types of inputs. For example, the authors point out the GLADE does not perform well when inferring the grammar of PDF objects. This is because PDF objects are flat and contain a large variety of content types while GLADE is better at learning the hierarchical structures of inputs.

3.3.2 Proposed Solution

Godefroid et al. propose the use of machine learning to solve this problem via statistical learning. The authors use recurrent neural networks to learn a grammar because RNNs can also be used to generate inputs from the learned grammar. The authors test their solution using the highly structured input of PDF files by learning PDF objects. The PDF objects are then used to create PDF files and are tested with the PDF parser of the Microsoft Edge browser.

3.3.3 Explanation of Algorithm

A character-level RNN is used, meaning it learns the probabilities of one character given all characters that have come before it. The probabilities are taken into account when generating an input. The LSTM chosen has 2 hidden layers, where each layer has 128 hidden states. Before a RNN can be used, the characters must be encoded

into numeric sequences. Here, all characters are translated into their unique numeric representation.

The authors also present a multitude of ways to create an input using the learned grammar: NoSample, Sample, SampleSpace, and SampleFuzz.

NoSample generates the next character given a sequence by choosing the best fit (i.e., the one with the highest probability). While this method will create inputs that are mostly likely to be accepted by the target program, it will always create the same input each time, therefore making it useless for fuzzing purposes.

Sample chooses the next character based on its learned probabilities, similar to choosing an expansion rule from a PCFG. Think of a LSTM as a function where the inputs are the previous characters and previous hidden states. The output of the LSTM function is a probability distribution, where each character has its own probability attached to it. A simplified example output would be the letter A having a probability of 95% and the letter B having a probability of 5%. Sampling picks the next character based on their probability (e.g., A has a 95% chance of being chosen).

SampleSpace is a combination of NoSample and Sample. It uses the NoSample strategy until the input ends with a whitespace. If the sequence ends with a whitespace, then Sample is used to choose the next character.

SampleFuzz is an algorithm introduced by Godefroid et al. to create inputs using Sample while also fuzzing them. Algorithm 1 shows how SampleFuzz works. First, it chooses the next character, *next_char*, using the Sample method. The probability of the chosen character is also obtained, represented as *next_char_prob*. A random number between 0 and 1 is created and labeled p_{fuzz}. The threshold numbers t_{fuzz} and p_t are parameters that determine if *next_char* will be changed into another character or not. If p_{fuzz} is greater than t_{fuzz} and *next_char_prob* is greater than p_t, then

next_char will be changed to the character with the lowest probability, obtained by the $argmin(prob\_dist)$ function.

Algorithm 1: SampleFuzz($prev\_char,\ prev\_hidden\_states,\ t_{fuzz},\ p_t$)

$prob\_dist \leftarrow ML\_Model(prev\_char,\ prev\_hidden\_states)$;
$next\_char,\ next\_char\_prob \leftarrow Sample(prob\_dist)$;
$p_{fuzz} \leftarrow random(0, 1)$;
if $p_{fuzz} > t_{fuzz}\ \wedge\ next\_char\_prob > p_t$ **then**
  $next\_char \leftarrow argmin(prob\_dist)$;
end
return *next_char*;

Each of the different methods of generating inputs are used for experimentation in fuzzing effectiveness.

3.4 DeepSmith

DeepSmith, proposed by Cummins et al., is a machine learning approach to compiler fuzzing [10]. In essence, it is Learn&Fuzz but for compilers instead. Cummins et al. used a LSTM to learn from handwritten programs to generate inputs for OpenCL compilers. Unlike Learn&Fuzz, DeepSmith uses a hybrid character-level and token-level encoder. Here, the OpenCL programming language's keywords and common names are treated as tokens. The remaining text is encoded at the character level.

Cummins et al. also used differential testing, where the generated inputs are tested using multiple programs. When determining the outcome of an input, a $\frac{2}{3}$ majority is used. This is used to detect anomalous behavior between the different programs.

3.5 Coverage-Guided Learning-Assisted Grammar-Based Fuzzing

Learn&Fuzz solves one of the main problems of grammar-based fuzzing, the need to manually create a grammar. However, it still has some problems that need to be addressed. Jitsunari et al. points to its limitations and proposed a solution that uses coverage-guided methods to improve upon it [26].

3.5.1 Current Solutions and Limitations

Jitsunari et al. point to two main limitations with the current work of grammar-based fuzzers. The first limitation has been pointed out repeatedly; it is the fact that grammar-based fuzzing requires an input grammar that must be created manually. The second limitation is the low code coverage, as generated inputs are not guaranteed to execute unexplored parts of the target program. The authors point to the fact that only a small subset of the grammar can contribute to better coverage.

The authors mention Learn&Fuzz and how it address the first limitation. But Learn&Fuzz has its own problems. The first issue is that it cannot generate instruction sequences, a more complex input. Instruction sequences also appear often in structured inputs, making this a major drawback. The second issue of Learn&Fuzz is shared with general grammar-based fuzzing, low code coverage.

3.5.2 Proposed Solution

Jitsunari et al. proposes the use of coverage information to continuously improve their RNN model. A general overview of their algorithm is shown in Algorithm 2. First, the RNN model is trained using sample inputs. Afterwards, the following is done in a loop for num_FTs iterations. The RNN is used to generate inputs and each

of the inputs is given a score according to the $CoverageScore(input)$ function shown in Algorithm 4. The top k scoring inputs are chosen to refine the model and the loop continues.

Algorithm 2: Cov_Guided_Training($num\_FTs,\ k$)

```
ML_Model.train(sample_inputs);
for (i ← 0; i < num_FTs; i ← i + 1) do
    inputs ← generate_inputs(ML_model);
    top_k_inputs ← get_top_k_inputs(inputs, k);
    ML_Model.train(top_k_inputs);
end
```

Like with Learn&Fuzz, this solution is testing by fuzzing PDF parsers. But instead of learning PDF objects, the RNN is used to learn PDF streams instead.

3.5.3 Explanation of Algorithm

Since the main algorithm itself has already been explained, this subsection will be used to explain the specifics of input generation and calculations of the scores for each input.

3.5.3.1 Input Generation

When using the RNN to generate inputs, the authors use the SampleFuzz method presented in Learn&Fuzz. The input is first generated using the Sample method and then fuzzed. The fuzzing method is shown in Algorithm 3.

By creating anomalies at the instruction level, the authors hoped to increase code coverage.

Algorithm 3: Cov_Guided_Fuzz($generated\_input,\ t_{swap}$)

```
foreach line ∈ generated_input do
    p_swap ← random(0, 1);
    if p_swap > t_swap then
        SwapLines(line.previous(), line);
    end
end
```

3.5.3.2 Code Coverage to Score Calculation

After the inputs are generated and tested with the target program, the code coverage metrics are converted into a score. Algorithm 4 shows how the scores are calculated. For each line that was executed, the score is increased by 100 if the line contains a branch (e.g., an if-statement). Otherwise, the score is only increased by 1. By calculating the score to favor branches, Jitsunari et al. believes it will create interesting inputs as branches are entrances to unexplored pieces of code.

Algorithm 4: Coverage_Score(*input*)

```
score ← 0;
Parser.parse_input(input);
foreach executed_line ∈ Parser.executed_lines() do
    if executed_line.hasBranch() then
        score ← score + 100;
    else
        score ← score + 1;
    end
end
return score;
```

3.6 Publicly Available Fuzzers

This section is meant to give the reader publicly available fuzzing resources. The main focus of this section are grammar-based fuzzers. Two other fuzzers are also mentioned, as they have been used to compare against proposed solutions [7, 42].

3.6.1 Grammar-based Fuzzers

The following fuzzers are grammar-based fuzzers. The majority of them are grammar-based fuzzers built for specific target programs. The "gramfuzz" library is also mentioned as it can be used to manually create a grammar for grammar-based fuzzing.

3.6.1.1 GLADE

GLADE, as mentioned before, uses examples of valid inputs and synthesizes a grammar using those inputs. Its implementation is written in Java and requires Java 1.7 and above. The GitHub repository comes with the source code as well as example inputs for the following programs: GNU sed, GNU grep, flex, XML, and Python [6].

In order to use GLADE to learn a grammar for another program and use the grammar for fuzzing, the user will need to use the compiled GLADE JAR file as a library. The GitHub page contains instructions on how to do so and the repository contains an example on using GLADE as a library.

3.6.1.2 gramfuzz

Gramfuzz is a Python library that you can use to build a grammar (CFG or PCFG) and then use it to generate inputs for whatever program you are testing [27].

3.6.1.3 CSmith

Csmith, as mentioned before, is a state-of-the-art generator for C programs to test C compilers [43].

3.6.1.4 CLSMith

This fuzzer is essentially CSmith but for OpenCL [32]. CLSmith was used to compare against DeepSmith [10].

3.6.1.5 Grammarinator

The fuzzer uses publicly available grammars for ANTLR and builds input test cases with them [23].

3.6.1.6 Domato

Domato is a grammar-based fuzzer for Document Object Model (DOM) [18]. The GitHub repo contains handwritten grammars in text files that is then parsed by a custom Python file and used to generate HTML files. The generator and grammar parser files can also be used as libraries, allowing it to be used for grammar-based fuzzing of other grammars.

The GitHub page also showcases a number of bugs found using in web browsers using Domato.

3.6.2 Other Fuzzers

The fuzzers below are popular ones and used as state-of-the-art fuzzers to be compared against.

3.6.2.1 Google's American Fuzzy Lop (AFL)

Originally developed by Michal Zalewski, Google's AFL uses a brute-force method along with an instrumentation-guided genetic algorithm [17]. The user supplies initial test cases and AFL mutates the test cases using traditional fuzzing strategies. And if the mutation improves coverage, then it is also used as a test case to be mutated. AFL was compared to GLADE [7].

AFL++ is a fork of Google's AFL and claims to be superior to Google's AFL [3]. It boasts more features, being faster than AFL, more and better mutations, and etc.

3.6.2.2 Radamsa

Radamsa is a blackbox fuzzer that operates by taking in valid inputs and modifying them [21]. During my research, this was the only explanation that was found of how this fuzzer worked. Radamsa was compared against REINAM [42].

Chapter 4

COMPARISON OF APPROACHES: BLACKBOX GRAMMAR-BASED FUZZING

In this chapter, I compare two approaches to grammar-based fuzzing: the use of grammar inference algorithms (GLADE) and the use of machine learning techniques (Learn&Fuzz). The code of REINAM was unavailable as it was undergoing refactoring. Coverage-Guided required a program that could give line-by-line coverage results, which I did not have access to at the time. In addition, both Coverage-Guided and DeepSmith use a hybrid character-token LSTM. This requires some knowledge about the input types to know which strings should be tokens. However, I only had limited knowledge of the inputs. Because of these limitations, only GLADE and Learn&Fuzz were tested. As both of these approaches are blackbox in nature, they are also compared to another open-source blackbox fuzzer, Radamsa.

4.1 Metrics

This section will discuss two types of metrics that will be used to measure the effectiveness of the grammar-based fuzzers. One set of metrics will be metrics for fuzzers and the other set of metrics will be to measure the performance when it comes to grammar inference.

4.1.1 Fuzzing Metrics

4.1.1.1 Coverage

Coverage is a measure of how much of the source code was executed by test inputs. There are a number of ways to measure coverage. For example, Godefroid el al. measured the number of unique instructions executed in DLL files [15].

Coverage can also be measured as the percentage of lines of code executed (line coverage) [25], percentage of statements executed (statement coverage) [36], and percentage of branch paths executed (branch coverage) [25]. Statement and line coverage are considered different because a single line of code may contain multiple statements.

This is an important metric because the goal of fuzzing is to find potential problems by exhaustively running all execution paths [20]. Unexplored code has the possibility of containing bugs that may lead to a security vulnerability.

For my testing, I used statement and branch coverage. These metrics were obtained and combined into a single coverage percentage using the Python *coverage* tool [8].

4.1.1.2 Number of Unique Bugs Uncovered

Finding bugs is the main objective of fuzzing [29]. However, that is not to say that a program is guaranteed to have bugs that could be found. For example, Godefroid et al. did not find any bugs when fuzzing the Microsoft Edge PDF parser [15]. The program was fuzzed for months beforehand, so it made sense that there were no bugs found [15].

When there are programs with many bugs that can be found, then it can be used as a metric for comparisons against the state-of-the-art methods. For example, Cummins et al. tested their proposed solution, DeepSmith, against CLSmith by comparing the number of defects that occurred during testing [10]. Cummins et al. also used the metric of *distinct* defects [10]. A fuzzer that finds the same bug multiple times is not as helpful as a fuzzer that can find multiple unique bugs. In addition, multiple inputs can trigger the same bug, so only counting the number of times a bug was encountered can be misleading [29].

For my testing, this metric is unused as I will be testing Python libraries. The libraries have already gone through bug testing, so no bugs were found.

4.1.2 Grammar Inference Metrics

There are two different types of uses for grammar inference algorithms. The first one is to infer a grammar. In this use case, the metric of pass rate is used. Pass rate, or precision, is the percentage of inputs created by the grammar that were accepted by the target program [7, 15]. This can be used to measure how much the inferred grammar is overgeneralizing.

To measure how much the grammar is undergeneralizing, recall is used. Bastani et al. and Wu et al. measured recall by having the true grammar create inputs and testing what percentage of them could be created by the inferred grammar [7, 42]. However, recall requires that the true input grammar to have already been created. Bastani et al. also use precision and recall to calculate the F1 score.

For my experiments, only pass rate is used. Recall requires having the input grammar to have been created, which was not accessible to me at the time.

The second use is to infer a grammar for grammar-based fuzzing. For this usage, the fuzzing metrics are used here. But as Godefroid et al. pointed out, better inference and learning does not necessarily mean better fuzzing [15].

4.2 Testing Implementation

This section will explain the details in using Learn&Fuzz and GLADE for comparisons. The comparisons used two input types: PDF objects and XML. Godefroid et al. pointed out that GLADE was not well suited for inferring the grammar of PDF objects as it is relatively flat with a lot of variety [15]. XML was also chosen as it had an hierarchical structure to it. In addition GLADE was able to reach almost perfect precision and recall with XML [7].

When using machine learning to create inputs, each method created 1,024 inputs. GLADE has three different types of input generators in its code. To give each a fair chance, 500 inputs were created using each generator, resulting in GLADE having 1,500 inputs.

When GLADE was used to create PDF objects, a total of 3,000 PDF objects were created. Explained later, GLADE used two different inferred grammars to create PDF objects and so each input generator created 500 PDF objects per inferred grammar.

The PDF objects were tested by attaching each PDF object to its own host PDF file and parsed with the Python *PyPDF2* library. In order to parse XML inputs, two different Python libraries were used: *xml.etree* and *xml.dom*. This was done because the coverage results of *xml.etree* were the same for GLADE and Learn&Fuzz. An additional library, *xml.dom*, was used to make sure it was not a fluke.

The dataset for PDF objects came from [45]. In addition, the XML dataset to train the ML model also came from there.

The datasets for PDF objects had two types, a small and a large one; each of them consisted of three datasets: a training dataset, a validation dataset, and a testing dataset. For our testing, I used the small dataset, which was 95,422 PDF objects in total. I also used code from the GitHub repository, *pdf_file_incremental_update_6.py*, and modified it to insert only one PDF object into a host PDF file (*host1.pdf*). However, I noticed that the *PyPDF2* library would only throw an error if the PDF object was inserted with an ID of 2. If inserted with any other ID, then *PyPDF2* would not throw an error no matter what. But this also caused many valid PDF objects to become invalid. With no other choice, I took all PDF objects of the small dataset and filtered them by inserting each PDF object into a host file and checking if *PyPDF2* would throw an error or not. At the end, there were 58,112 PDF objects. The filtered PDF objects were anywhere from 14 to 496 characters in length.

There is only one XML dataset, consisting of four large XML files. The files range from about 500 to 33,000 lines of XML.

4.2.1 PDF Objects and XML

An example of a PDF object is in Figure 4.1. A PDF object starts with "obj" and ends with "endobj". The "<<" and ">>" denote a dictionary structure, where the key is followed by their value [15]. A PDF object can also have array objects, with information being held inside the square brackets. Not shown in the figure is the fact that PDF objects inside PDF files also have a two numbers before the "obj", an identifier and a generation number. The generation number is incremented if the object was replaced by a newer version [15]. An example would be "2 0 obj" followed

```
obj
<<
  /Type /Annot
  /Subtype /Link
  /Border [ 0 0 0 ]
  /H /I
  /C [ 1 0 0 ]
  /Rect [ 320.841 392.952 336.66 402.736 ]
  /A <<
    /S /goTo
    /D 323 0 R
  >>
>>
endobj
```

Figure 4.1: Example of a PDF object.

by the rest of the object. The dataset of PDF objects did not contain these numbers as they were added when the PDF objects were inserted into a host PDF file.

Three examples of XML are in Figure 4.2. A XML element consists of a starting tag, an ending tag, and anything in between the two tags. The tags can be named anything, but the start and end tags must match (tags are also case sensitive). Example 1 shows that there can be elements within other elements. An XML element can be empty if there is nothing in between the start and ending tags, or can be denoted by a "/", as shown in Example 2. XML elements can also have attributes. Example 3 gives an example of this, where the *example* element contains the *text* attribute (which has the value of "Sample Text"). The value of an attribute must be in quotes.

4.2.2 Learn&Fuzz

For Learn&Fuzz, the machine learning model was recreated according to what Godefroid et al. used: a 2-layer LSTM with 128 neurons per layer [15]. There were some

```
EXAMPLE #1:
<example>
    <text> Sample Text </text>
</example>

EXAMPLE #2:
<example/>

EXAMPLE #3:
<example text="Sample Text"/>
```

Figure 4.2: Three examples of XML.

differences to reduce the training time, such as training with a batch size of 32. My training set consisted of 58,112 PDF objects, close to the 63,000 number in the paper.

The machine learning model was trained in the following way: for the i-th iteration, the input was $text[i*d : (i+1)*d]$ and the output was $\text{text}[(i*d)+1 : ((i+1)*d)+1]$, where $text[start\_index : end\_index]$ is a substring of $text$ between indices $start\_index$ to end_index, inclusively and exclusively, respectively. Learn&Fuzz used *d=100*, so that was what I used as well.

Godefroid et al., when reporting their results, mainly used 40 epochs. As such, I also trained the model with 40 epochs. And to create inputs, I used the following methods: Sample, SampleSpace, and SampleFuzz (see Section 3.3.3 for details). NoSample was skipped as that would only create the same input.

The PDF objects were concatenated and put into a single file, following how Godefroid et al. trained the model.

The XML dataset consisted of 4 large XML files. Rather than combine them, the four files were used independently. Whitespace was added to the end of each file to ensure the ending portions would be used for training.

4.2.3 GLADE

In order to use GLADE to infer a custom grammar, it had to be used as a library. Unlike with machine learning, there weren't any parameters to deal with. The main steps were to supply GLADE with a few sample inputs and to create a *DiscriminativeOracle* class. The coding and input creation parts will be explained more in Section 4.4.3 and Section 4.4.4, respectively.

For PDF objects, GLADE inferred the grammar using only 2 PDF object examples due to time constraints as inferring from a single example took hours. They were inferred independently and while GLADE does have the option to merge grammars after inference, it resulted in errors when using the merged grammar for fuzzing. I believe this was because I used the merging code to save the inferred grammars, resulting in a doubly-merged grammar that could not be used for fuzzing. Due to time constraints, this mistake could not be fixed and the inferred grammars had to be used for input generation independently.

GLADE came with its own XML examples. These examples were used to infer the XML grammar.

4.2.4 Radamsa

Radamsa is an open-source blackbox fuzzer that uses examples given to it and creates new inputs based on the examples [21]. For both PDF objects and XML, it only used two examples to generate inputs. Each example was used to generate 1,000 new inputs.

For PDF objects, it used the same two examples as GLADE did. And for XML, it used two example XML inputs that came with GLADE.

Table 4.1: Approximate grammar inference times.

Method	Inference Time
Learn&Fuzz: PDF Objects	3 Hours
Learn&Fuzz: XML	2 Hours
GLADE: PDF Object 1 (Local Machine)	3 Hours
GLADE: PDF Object 2 (Google Colab)	8 Hours
GLADE: XML (Local Machine)	7 Minutes

4.3 Comparison: Grammar Inference Time

This section will be comparing the two grammar-based fuzzing methods on the time to infer a grammar. Table 4.1 shows the results of the times. The subsections below will give details.

4.3.1 Learn&Fuzz

The time it takes a machine learning model to learn the syntax depends on a few variables, with the main ones being the amount of data, number of epochs, batch size, and hardware acceleration.

Using stochastic gradient descent (weights of the model are updated for every piece of data) on a large dataset would take a lot of time, which is where batch stochastic gradient descent comes in. This splits up a dataset into batches of size n, where the machine learning model updates its weights after going through n examples.

The number of epochs is how many times the machine learning model goes through the entire dataset. The higher number of epochs, the more the ML model is able to fit to the examples given to it. As mentioned before, the model was trained with 40 epochs. On Google Colab, 40 epochs with a batch size of 32 took about 3-4 hours with GPU acceleration for PDF objects and around 2 hours for XML.

4.3.2 GLADE

For GLADE, grammar inference takes a variable amount of time, depending on the length and complexity of its seed inputs as well as the system it is running on. For PDF objects, inferring from a single seed input took about 3 hours on a system with an Intel i7-10750H CPU (with maximum speed set to 3.5 GHz). GLADE was also tested on Google Colab, where it took 8 hours. And using the seven small XML examples that came with GLADE took around 7 minutes in total.

The XML examples of GLADE had 158 characters in total. The first PDF object (local machine) was 175 characters and the second PDF object (Google Colab) was 309 characters in length. From personal experience, inferring from a simple Python program such as "print("Hello World!")" took much longer than inferring from XML.

4.4 Comparison: Ease of Use

This section is to compare how easy or hard it was to use machine learning and GLADE when it comes to the coding aspect and using them to create inputs.

4.4.1 Learn&Fuzz: Code

You have to know how to use machine learning methods, which include preprocessing the data, splitting the data into how you want to learn it, splitting into batches, shuffling it, etc. There may also be some confusion on how to create a model and use it to create inputs. For example, logits vs. softmax activation function, returning states and/or returning sequences, and other parameters.

4.4.2 Learn&Fuzz: Input Creation

The main difficulty of using machine learning to create inputs is that the model does not *really* learn a grammar. That is, it only learns the probability distribution of the next characters based on previous inputs. In order to create the input, the user needs to two things: a starting and an ending sequence. The starting sequence is the first input that starts the creation process. I then had the model choose the next character based on the selected algorithm until the input ended with the ending sequence. As an example, all PDF objects start with "obj" and end with "endobj", making it easy to create inputs without much problem.

An example of an input that requires more knowledge to create would be XML. XML expects its starting and ending tag to be the same, such as *<member> sample text </member>*.

Any other tags inside will also need to be the same. For example,
<member> <summary> sample text </summary> </member>

Because the *<member>* and *</member>* tags appeared often in the dataset used to train the model, it was used as the starting and ending sequences, respectively.

When creating the input, there may be a possibility where the program runs forever because it is not able to reach the ending sequence. Using the Sample method, this problem does not appear to be an issue. The SampleFuzz algorithm, on the other hand, may have this problem. It becomes hard to tell if the program will eventually reach the ending sequence or not given enough time. This problem is apparent when creating PDF objects, but it has been observed to eventually reach the ending sequence. When creating XML inputs, it seems to run forever.

The hypothetical reason for this is the fact that SampleFuzz ($t_{fuzz} = 0.9$, $p_t = 0.9$) is constantly modifying the input as it is created. There is a ten percent chance for the input to be fuzzed, but it is only modified if the probability for the next chosen character is higher than ninety percent. If chosen to be modified, then the next character is changed into the one with the lowest probability. Through observations, it would seem this messes up the input creation process and causes the XML ending tag to be unreachable (observed for about 15 minutes).

One solution used is to short circuit the ending sequence. For example, it was observed that the XML creation process would reach $</m$ only for the next character to be modified and shifting it away from the full ending sequence. So, $</m$ became the ending sequence instead and *ember>* was filled in when storing the inputs into a text file. For testing, the ending tag $</$ was also used, with *member>* being appended later.

The SampleSpace method also had a long wait time when creating XML objects. When using the full ending sequence, *</member>*, it took about 2 hours to create 1,024 inputs.

Table 4.2 shows the starting and ending sequence of each method and input type. Table 4.3 shows the approximate time of each input creation method takes to create

Table 4.2: Start and end sequences for Learn&Fuzz input creation methods.

Input Type	Start Sequence	Sample and SampleSpace End Sequence	SampleFuzz End Sequence
PDF Object	obj	endobj	endobj
XML	<member	</member>	</ </m

Table 4.3: Approximate times for each Learn&Fuzz method to create 1,024 inputs.

Input Type	Sample	SampleSpace	SampleFuzz ($t_{fuzz} = 0.9$, $p_t = 0.9$)
PDF Object	5-30 minutes	15 minutes	15-30 minutes
XML	30 minutes	2 hours	</: 15 minutes </m: 2 hours

1,024 inputs. Time may vary for others based on implementation and the fact that inputs are created using probabilities.

4.4.3 GLADE: Code

The knowledge of the Java programming language is needed in order to use GLADE for custom grammar inference.

In order for GLADE to infer a grammar, the user will need to create a class for the *DiscriminativeOracle* interface that implements the *query(String)* method. The *query* method is used to determine if the inputs created by GLADE are valid to the target program or not, where an input created by GLADE is the String parameter to the method. Therefore, the user will need this method to call the target program,

have the target program read the input, and return a Boolean representing whether the input was valid or not.

GLADE will continuously create inputs and test them with the target program to infer the grammar. After the grammar has been inferred, the grammar will need to be saved using the builtin methods.

4.4.4 GLADE: Input Creation

To create inputs for the target program with GLADE, the saved grammar file will be loaded in and be used to create inputs with. Unlike machine learning techniques, there is no need to have starting and ending sequences. The user only needs to load in the grammar use it to create one of three sampler classes: *GrammarSampler*, *GrammarMutationSampler*, and *CombinedMutationSampler*.

GrammarSampler is the uniform sampling method and *GrammarMutationSampler* is the grammar-based fuzzing method explained in Section 3.1.4. *CombinedMutation-Sampler* uses the grammar-based fuzzer to create the input and has an even chance of either returning the input as it is or modifying it. The input is modified n times, where n is a number given by the user. For each modification, a random character of the input is chosen and that character has an even chance of either being replaced by a randomly generated character or deleted.

Compared to machine learning methods, input creation using GLADE is much simpler as there is no need to fear the possibility of the program running forever. It is also much faster. Sample is much faster than SampleFuzz when using ML, but it can still take five to thirty minutes and even longer to create 1,024 inputs, depending on how fast the creation process can reach their ending sequence. GLADE, on the other hand, took only seconds to create 1,500 inputs.

Table 4.4: PDF and XML Fuzzing Results

Input Creation Method	PyPDF2 Coverage	PyPDF2 Pass Rate	XML Coverage	XML Pass Rate
Radamsa	#1: 27.49% #2: 27.83% ALL: 27.83%	#1: 49% #2: 53.3% ALL: 51%	14.98% 15.48%	12.5%
SampleSpace	25.16%	33.6%	15.05% 15.48%	78%
Sample	27.62%	52.246%	15.05% 15.48%	14.55%
SampleFuzz 90_90	27.49%	4.3%	14.98% 14.98% 15.48% 15.48%	“</m”: 0.3% “</”: 8.6%
SampleFuzz 95_90	27.53%	8.5%	N/A	N/A
SampleFuzz 95_95	27.62%	10.35%	N/A	N/A
SampleFuzz 99_90	27.7%	30%	N/A	N/A
SampleFuzz 99_95	27.7%	29.6%	N/A	N/A
GLADE	#1: 27.55% #2: 27.73% ALL: 28.07%	#1: 63% #2: 64.4% ALL: 63.77%	15.05% 15.48%	66.1%

4.5 Comparison: Results

This section will compare machine learning and GLADE when it comes to learning the grammar (pass rate) and their effectiveness when used for fuzzing purposes (coverage).

Table 4.4 shows the results of Learn&Fuzz and GLADE when it comes to fuzzing PDF objects and XML. SampleFuzz is labeled using SampleFuzz (t_{fuzz})_(p_t). For example, SampleFuzz 90_90 is when $t_{fuzz} = 0.9$ and $p_t = 0.9$.

XML coverage contains two parts: the coverage when parsing XML from strings and the coverage when parsing XML files. SampleFuzz 90_90 contains four coverage results as there were two different methods used to create the inputs. As mentioned in Section 4.4.2, the ending tags $</m$ and $</$ were used with the rest of the ending sequence being added afterwards. The results of using the $</m$ tag is above the $</$. For the other variations of SampleFuzz, there were no XML experiments done due to time constraints.

For GLADE's PDF results, it used two examples and the two inferred grammars were used to produce inputs. As seen in Table, 4.4, the results of first grammar is #1 and the results of the second grammar is #2. Combining the inputs created by the two grammars gives us the results recorded by "ALL".

The Python libraries used for testing contain a large number of methods that a user can use to modify the parsed input. As the fuzzers are only meant to test the parsing part and information extraction parts of these programs, the low code coverage is to be expected.

From a look at the results, GLADE is better when it comes to pass rate and coverage.

4.5.1 Learn&Fuzz Results

With the machine learning methods, one thing that surprised me was the fact I did not get the high pass rates reported by Godefroid et al. (e.g., 68.24% with SampleFuzz) for creating PDf objects [15]. Instead, I got the abysmal pass rate of 4.3% when using the SampleFuzz method using the same variables (SampleFuzz 90_90). When the t_{fuzz} variable is higher, there is a lower chance of the input being fuzzed as its created. As seen in Table 4.4, this produces a higher pass rate and coverage for PyPDF2. And

```
obj
o(jPDDlvical Cod saror reQuirertoWity Object in A
```

Figure 4.3: SampleFuzz confusing the ML model. The open parenthesis is the fuzzed character.

```
obj
<<i
  /P 233 0 R
a
endobj
```

Figure 4.4: SampleFuzz causing the input to be invalid (*i* and *a* are the fuzzed characters).

while Sample did have a higher pass rate, it also did not reach the pass rate reported by Godefroid et al.

A main reason for the lowered pass rate was that Sample created PDF objects which had the starting sequence repeated, which may have been the consequence of concatenating the PDF objects for training. When the repeating sequence was removed for the inputs that had them, the pass rate increased to 69%. The coverage, however, remained the same.

But for SampleFuzz, the main reason for the low pass rate is that the fuzzing algorithm frequently fuzzes important sequences, causing the input to become invalid. Figures 4.3 and 4.4 are examples of this, where the fuzzed characters are in locations that cause the input to become invalid. Figure 4.3 also shows a reason why the input generation for SampleFuzz takes a long time, where the model becomes confused after the next character was fuzzed. SampleFuzz also liked to fuzz characters in the ending sequence, resulting in invalid inputs and even longer generation times.

As seen in Table 4.4, higher numbers for t_{fuzz} and p_t mean less fuzzing, which in turn means higher pass rate and coverage.

What is also surprising is the low pass rate of SampleSpace for PDF objects. Godefroid et al. had SampleSpace always near 100% pass rate at any number of epochs. When looking at the inputs by eye, they look to be mostly valid PDF objects with a few invalid outliers.

One big reason for this is because SampleSpace causes the formed PDF object to repeat their values (e.g., something like "/Type /Font" is repeated a few lines down). This repetition causes the created PDF object to become invalid. If this repetition problem occurred less often, then the pass rate could be much higher. The lower code coverage of SampleSpace was to be expected, as the algorithm does not attempt to create interesting inputs. For example, the created inputs always have "/Type /Font" as the first key-value pair.

While it had trouble with PDF objects, SampleSpace had less of a problem when it came to XML.

For XML results, there is not much of a difference when it comes to coverage except for SampleFuzz. Weirdly, SampleFuzz had a lower code coverage than Sample and GLADE when parsing XML from strings but then had the same code coverage when parsed from a XML file. SampleFuzz having a low pass rate was expected, but Sample also struggled to have a high pass rate. The main reason was because XML inputs generated using Sample had starting and ending tags that did not match. For example, "<summary>" would have the ending tag of "</returns>". It seems the model learned that XML requires a start and end tag, but it didn't learn that the tags need to match. Figure 4.5 shows an example of this.

```
<member name="P:System.Drawing.Printing.Page.Unit">
    <summary>
        A system-defined
        <see cref="T:System.Drawing.Pen" />
        object set to a system-defined color.
    </returns>
    <param name="style">
        The
        <see cref="T:System.Drawing.Pen" />.
    </returns>
    <param name="caltType">
        A
        <see cref="T:System.Drawing.Printing.PaperKind.HatchStyle" />
        methods.
    </summary>
</member>
```

Figure 4.5: Example of a XML input created using Sample.

Table 4.5: GLADE PDF object fuzzing results.

GLADE Input Creation	PyPDF2 Coverage	PyPDF2 Pass Rate
1-sample	26.53%	65%
1-fuzzer	26.44%	79.2%
1-combined	27.38%	45.2%
1-all	27.55%	63%
2-sample	27.21%	65.8%
2-fuzzer	27.38%	81.6%
2-combined	27.64%	45.8%
2-all	27.73%	64.4%
ALL	28.07%	63.77%

4.5.2 GLADE Results

With GLADE, it had a higher pass rate than any of the input creation methods of Learn&Fuzz. In addition, the results show GLADE being able to match Sample and SampleFuzz in terms of code coverage with inputs created from one inferred grammar. When the inputs are combined, they result in the highest code coverage of 28.07%.

GLADE, using only two examples, was not able to infer the large variety of PDF objects. However, its higher code coverage along with higher pass rate may point to two things: 1) GLADE was able to infer new or different inputs and/or 2) GLADE's grammar-based fuzzers created more interesting inputs. As mentioned in Section 3.1.3, GLADE infers a grammar by creating inputs and checking if the target program accepts or rejects the inputs. As such, it is no surprise that GLADE has such a high pass rate. This process may also be the reason GLADE has a higher code coverage with PDF objects, as it was able to explore different input arrangements.

Table 4.5 show the individual coverage and pass rate results of each creation method and inferred grammar. Sample uses the grammar to create an input through uniform sampling, fuzzer uses the grammar to modify an example input, and combined mixes in invalid inputs on purpose. From the results, it was surprising to see that the grammar-based fuzzer had a higher pass rate than the sampler. However, their coverage is about the same, with either one being higher depending on the grammar and/or inputs generated. The combined method consistently has the highest coverage. One interesting result is the fact that coverage increases when combined the different methods together. This means that each input creation method and grammar was able to execute different lines of code, making GLADE an effective fuzzer.

4.5.3 Comparison to Radamsa

For PDF objects, Radamsa performed surprisingly well. It managed to outperform most of the ML methods when it came to coverage and pass rate. However, GLADE was still able to come out on top. This might be because this PDF objects are flat with a lot of variety (i.e., not much structure to it). Because of this, Radamsa was able to obtain great results.

When tested on a input with more structure (XML), both ML methods and Radamsa have very low pass rates, expect for SampleSpace, whose pass rate surpassed GLADE's. But one thing to remember is that GLADE's generated inputs also contain ones that were semi-valid and invalid on purpose. Radamsa also shares the oddly lowered coverage when parsed from a string. Sadly, the coverage results of XML are not interesting and have almost the same numbers each time. This might be because while XML is an input type that has some structure to it, it is still a relatively simple input.

Given a more complex structured input, I would hypothesize that GLADE would be the clear winner. As the results for PDF objects show, there is a slight correlation between pass rate and coverage, as higher pass rate would mean that the input was not outright rejected by the parser and thus able to execute more lines of code.

Due to time constraints and lack of data for such an input type, this hypothesis was not able to be fully tested. Future work could be done to expand upon these experiments.

Chapter 5

PROPOSED IMPROVEMENTS

In this chapter I propose two improvements to the blackbox grammar-based methods I tested with: improving the user-friendliness of GLADE through the use of a configuration file and improving machine learning methods through supplementary inputs from GLADE. The first proposed improvement has a prototype implementation. Due to time constraints, the second proposed improvement has not been implemented.

5.1 Improvements to User-Friendliness

This section proposes a method to improve GLADE's usability via abstractions. Currently, GLADE must be used as a Java library in order for it to learn the grammar of other inputs. While GLADE does come with an example file of how to do so, it still requires the user to research how to save the grammar and how to use the grammar for fuzzing. The proposed abstractions are to create a file that the user can call to learn and fuzz any inputs they desire and to use a configuration file to eliminate the need to interact with the code.

5.1.1 Problem Statement and Proposed Solution

GLADE is available to anyone to use for learning a grammar for a target program for grammar-based fuzzing. However, it is not too user-friendly. The main issue is the fact that GLADE is only programmed to learn inputs and fuzz the programs that

were tested in the paper. In order to learn an input grammar for another program, the user has to use GLADE as a library.

But learning how to use GLADE takes time to understand the code and what each part does. GLADE has no documentation on how to use it as a library. It does contain example code of how to use it as one, but it only shows how to infer a simple grammar and then using the grammar for fuzzing. It does not show how to perform two important processes: how to call the target program and give it inputs and how to save the grammar. The only option then is to look through the source code, which contains no comments besides a software license.

My contribution will be abstracting away some of the difficulties that come with GLADE. Rather than needing the users to interact with the code, all they will need to do is make a configuration file. The configuration file should contain all necessary information. For example, where the example inputs are, where to save the grammar, how to call the target program via command line and give it inputs, and etc.

The syntax of my proposed configuration file is simple. It is a valid option name followed by an equal sign and the user-inputted value. Comments made to the configuration file are denoted by the pound sign, "#". Figure 5.1 shows an example of a configuration file. The options and example values are on the left and comments explaining the options are on the right. The file contains options that require a user-defined value and options that have default values. The default values were obtained from values used in GLADE's example code of how to use GLADE as a library.

Further work can be done to create a program to help users with choosing the values for options that may not be intuitive, such as choosing the either the input or error stream to determine if the target program ran without any problems.

```
learn_or_fuzz = fuzz                                # mode of operation (learn or fuzz)
target_program = python3                            # Which program to call in the command line of the OS
file_extension = .py                                # What type of files does the program use
input_or_error_stream = error                       # Use input or error stream to determine correct run
stream_empty? = yes                                 # Should the chosen stream be empty if run correctly
sample_input_folder = data/inputs-train/python3     # Where the sample inputs are stored
grammar_output_folder = data/grammars               # Where should the output serialized grammar be stored
save_CFG? = yes                                     # Save the CFG as text?
CFG_output_folder = data/grammars/python            # Where to save the CFG

#-------------New options------------------
#The options below can be ignored because they have default values
fuzz_type = combined                                 # grammar, mutation, or combined; mutation default
fuzz_output_num = 10                                 # How many fuzzing outputs to create (in fuzz mode)
log_file_name = log.txt                              # What should the created log file be named?
log_verbose? = yes                                   # Make the log file verbose?
random_seed = 1                                     # Integer seed for reproducible fuzzing outputs
sample_max_len = 1000                                # Maximum length of fuzzing output
num_mutations = 20                                   # Number of mutations for output
```

Figure 5.1: Example of a Configuration File to Run GLADE.

Another problem with GLADE is that it infers the grammar using each sample input and then merges all the inferred grammars together at the end. If a user wants to add to the inferred grammar using a new sample input, there is no option for them to do so. Work should be done to implement this option, either with the configuration file or in a separate program.

5.1.2 Implementation

An algorithm for implementing the proposed solution is described as follows.

Step 1. All default values are loaded for all options that have default values.

Step 2. The program loads the text file containing the configuration. The user can choose which text file to load in by providing the name of the text file as an argument to the program. If there is no argument passed, then the a default name is used.

Step 3. The configuration file is parsed; each option and their value are stored in memory. During this step, the user-inputted values will need checked for errors and that the syntax of the configuration file is correct.

Step 4. After the configuration file is read, GLADE is run using the user defined values.

I created a prototype implementation, which can be found at [41]. Currently, the "save_CFG?" option does nothing. Saving the CFG as a text file would allow the grammar to be used by other grammar-based fuzzers. In addition, there are some assumptions made, such as the target program not needing any command line options. Future work can be done on improving the options of the configuration file, finding a format to save the CFG as text, and implementing it.

5.2 Using Grammar Inference to Help Machine Learning Training

This section proposes the use of inputs generated by GLADE, or any grammar inference fuzzer, to supplement the training data of machine learning. When there is little to no data, GLADE-generated inputs may be used to train a model, with the assumption that the model will then be able to generate new and interesting inputs. And when there is enough data, GLADE-generated inputs can added. Real world data cannot contain all possible valid inputs and GLADE has been shown the generate interesting valid inputs. Due to time constraints, the proposed improvement has not yet been implemented.

5.2.1 Current Problems

As mentioned by Wu et al., machine learning (ML) approaches, given a large number of seed inputs, are able to learn a model representing an input grammar [42]. A big problem with this approach is that there may not be enough data to learn from. And even if there is enough data from real-world users, there may be some interesting valid inputs that were not created by the users. And this means that the ML model might not be able to generate interesting inputs useful for fuzzing.

To solve the above problems, I propose the solution of combining the use of the current state-of-the-art grammar inference algorithms with ML approaches. Currently, GLADE and REINAM are state-of-the-art. However, only GLADE is available to be publicly used. As such, the grammar inference algorithm to be used in this proposal will be GLADE.

5.2.2 Proposed Solution

My proposed solution, assuming a whitebox scenario, is to first use a whitebox test generator to generate test cases for the target program. The test cases will be used as sample inputs for both GLADE and the ML model. If in a blackbox scenario or there is no test generator available, then any examples can be used.

Once GLADE works through the sample inputs and generates a grammar, the grammar can then be used to generate data for the RNNs/LSTMs to learn. As explained by Godefroid et al., there are some types of inputs, such as PDF objects, that GLADE is not good at inferring [15]. By using a combination of input generation techniques, a variety of inputs can be created to train the ML model. This is done in hopes that

Figure 5.2: Example of a valid Python program created by GLADE.

the ML model can take the generated inputs and create even new inputs that can be used for fuzzing.

This should only be a temporary measure when a program does not have enough data to learn a ML model. After there is enough data, then GLADE can also learn from the new data to create supplementary data for the ML model to train on. The reason for this is because GLADE is able to generalize a grammar based on the sample inputs and may create new inputs that have not been seen in real-world uses. Because a ML model can only learn from its given inputs, using GLADE to supplement inputs may create better inputs for fuzzing.

One example of an input that can be supplemented by GLADE is in Figure 5.2. As you can see in figure, the editor for Python detects a problem with the program. However, when run with the command "python3 test.py", where *test.py* contains the program, there is no error thrown. Another example is when a program has the NUL unicode character as the beginning character of the program. These all are valid Python programs that real-world users may not have thought about creating.

5.2.3 Current Challenges

Due to time constraints, this proposed improvement has not been implemented. The time it takes for GLADE to infer a grammar for a single sample input is proportional to the sample input's complexity and length. This means that it may take a non-trivial amount of time to infer from a large number of real-world examples. In addition,

extensive testing will need to be done if GLADE does, in fact, help the ML model to create better fuzzing inputs at each stage of data availability.

Chapter 6

FUTURE WORK

6.1 Blackbox Grammar Inference

Currently, research into blackbox grammar inference has been limited. As mentioned by Wu et al., the precision and recall of REINAM was measured with ideal grammars and may not be complicated enough to reflect real-world grammars [42]. Bastani et al. also measured recall using relatively simple grammars, such as XML and URL [7]. More work should be done in researching how well the grammar inference algorithms work on complicated grammars.

6.2 Machine Learning Parameters

Godefroid et al. mentioned that there are many other RNN approaches and that theirs is by no means the best [15]. For example, machine learning techniques have many parameters to consider: batch size, number of epochs, dataset size, number of layers, number of neurons per layer, and etc. In addition, there is another well-known RNN architecture, the gated recurrent unit (GRU).

Future work could be done to fine-tune these parameters as well as explore how the different RNN architectures compare.

6.3 Pass Rate and Coverage

As mentioned in Section 4.5.3, there may be a correlation between pass rate and coverage. Godefroid et al. mentions there being a tension between coverage and pass rate, as the goal of learning is to maximize pass rate but the goal of fuzzing is to maximize coverage [15]. My results also suggest that coverage suffers if the pass rate is not high enough. Future work could be done to explore a good ratio between the two metrics.

Because of lack of data and time, I was not able to fully explore GLADE vs ML vs Radamsa fully. Given a much more complex and structured input, which grammar-based fuzzer is better? And how will a blackbox fuzzer such as Radamsa perform? And why?

In addition, work should also be done to measure the number of unique bugs encountered. As Klees et al. mentioned, fuzzers are used to find bugs in the target program [29]. And while there may be a correlation between coverage and number of bugs, that correlation may be weak [29]. Therefore, the number of unique bugs should be the primary metric.

Chapter 7

CONCLUSION

As seen in the results, while GLADE may not have been able to infer the many variations of PDF objects as machine learning methods can, it was still able to have a higher code coverage with only 2 examples. This may be because of two things: GLADE was able to infer grammar structures that machine learning couldn't and/or GLADE's input generators are more effective.

Machine learning methods are much more complicated than GLADE, as they require a large amount of data along with having many parameters to consider. In addition, ML methods can be significantly slower if you do not have a proper system with GPU acceleration.

When it comes to the use of machine learning for fuzzing, the Sample method is the best. While the idea behind SampleFuzz is sound and did work well for Godefroid et al., it did not work well for my experiments. Fuzzing characters with a high probability consistently caused the inputs to become invalid. The fuzzing algorithm can also throw the input generation off for a while before it finally reaches the ending sequence. It also liked to fuzz the ending sequence, which both caused it to become invalid and increased the input generation time even more.

Weirdly enough, the machine learning methods did not perform as well when it came to XML. Looking at some of the inputs created using Sample, we see that the inputs have the correct structure, but the starting and ending tags do not match. The model was able to learn the structure, but not the specific need for matching start and end tags.

As the pass rate of the SampleFuzz algorithm increases, so does its coverage. Because of this, there may be a correlation of pass rate and coverage when it comes to fuzzing. The correlation is broken with SampleSpace, where it has a high pass rate by low coverage. That is because SampleSpace purposefully plays safe in order to obtain a higher pass rate. However, higher pass rate does not mean higher coverage. GLADE's input method of purposefully adding in invalid inputs lowers the pass rate but increases the code coverage. A good fuzzer should have a good balance between pass rate and coverage.

From my experiences with the grammar-based fuzzers, I proposed improvements to them. I proposed an improvement to GLADE's user-friendliness through the use of a configuration file and an improvement to machine learning fuzzing through the use of supplementary inputs created by GLADE.

Further experiments will need to be done in order to fully explore the effectiveness of current research into automating grammar-based fuzzing, especially with a much more complex input that has a hierarchical structure to it. As seen with the XML test, SampleFuzz and Radamsa got a lower pass rate compared to Sample and GLADE. While XML does have a structure, it is still a relatively simple input so the coverage results were more or less the same across the board. With a more complicated input, I can only assume that it would follow the trend of needing a decent pass rate to obtain good coverage results.

BIBLIOGRAPHY

[1] Cal Poly Github. http://www.github.com/CalPoly.

[2] Metrics. https://www.cve.org/About/Metrics.

[3] AFLplusplus. Aflplusplus/aflplusplus, May 2022. https://github.com/AFLplusplus/AFLplusplus.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[5] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[6] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Obastani/glade, Sep 2016. https://github.com/obastani/glade.

[7] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 95–110, New York, NY, USA, 2017. Association for Computing Machinery.

[8] N. Batchelder. Nedbat/coveragepy. https://github.com/nedbat/coveragepy.

[9] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[10] C. Cummins, P. Petoumenos, A. Murray, and H. Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018,

page 95–105, New York, NY, USA, 2018. Association for Computing Machinery.

[11] M. Dawson, D. N. Burrell, E. Rahim, and S. Brewster. Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3(6):49–53, 2010.

[12] R. Fan and Y. Chang. Machine learning for black-box fuzzing of network protocols. In S. Qing, C. Mitchell, L. Chen, and D. Liu, editors, *Information and Communications Security*, pages 621–632, Cham, 2018. Springer International Publishing.

[13] L. Gaines. Cost of fixing vs. preventing bugs, Feb 2021.

[14] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber. Learning precise timing with lstm recurrent networks. *J. Mach. Learn. Res.*, 3(null):115–143, mar 2003.

[15] P. Godefroid, H. Peleg, and R. Singh. Learn amp;fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59, 2017.

[16] I. Gomes, P. Morgado, T. Gomes, and R. Moreira. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009.

[17] Google. Google/afl, Jun 2021. https://github.com/google/AFL.

[18] googleprojectzero. Googleprojectzero/domato, Sep 2021. https://github.com/googleprojectzero/domato.

[19] A. Graves. Generating sequences with recurrent neural networks, 2014.

[20] J. Heffley and P. Meunier. Can source code auditing software identify common vulnerabilities and be used to evaluate software security? In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*, pages 10 pp.–, 2004.

[21] A. Helin. Aki helin / radamsa, May 2022. https://gitlab.com/akihe/radamsa.

[22] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, nov 1997.

[23] R. Hodován. Renatahodovan/grammarinator, May 2022. https://github.com/renatahodovan/grammarinator.

[24] M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725, 2016.

[25] M. Ivanković, G. Petrović, R. Just, and G. Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 955–963, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Y. Jitsunari and Y. Arahori. Coverage-guided learning-assisted grammar-based fuzzing. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 275–280, 2019.

[27] J. Johnson. D0c-s4vage/gramfuzz, May 2020. https://github.com/d0c-s4vage/gramfuzz.

[28] D. Jurafsky and J. H. Martin. *Speech and Language Processing (3rd ed. draft).* 2021. https://web.stanford.edu/~jurafsky/slp3/.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. 2018.

[30] S. Lee, H. Han, S. K. Cha, and S. Son. Montage: A neural network language Model-Guided JavaScript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630. USENIX Association, Aug. 2020.

[31] J. A. Lewis, Z. M. Smith, and E. Lostri. The hidden costs of cybercrime, Jun 2022.

[32] C. Lidbury. Chrislidbury/clsmith, Apr 2017. https://github.com/ChrisLidbury/CLSmith.

[33] B. Liu, L. Shi, Z. Cai, and M. Li. Software vulnerability discovery techniques: A survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 152–156, 2012.

[34] X. Liu, X. Li, R. Prajapati, and D. Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):1044–1051, Jul. 2019.

[35] J. Oncina and P. Garcia. Identifying regular languages in polynomial time. In *ADVANCES IN STRUCTURAL AND SYNTACTIC PATTERN RECOGNITION, VOLUME 5 OF SERIES IN MACHINE PERCEPTION AND ARTIFICIAL INTELLIGENCE*, pages 99–108. World Scientific, 1992.

[36] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the*

Foundations of Software Engineering, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[37] J. Ponciano. 'extremely destructive' russian cyberattacks could cost u.s. billions of dollars in economic damage, goldman warns, Apr 2022.

[38] M. Sablotny, B. S. Jensen, and C. W. Johnson. Recurrent neural networks for fuzz testing web browsers. *Information Security and Cryptology – ICISC 2018*, page 354–370, 2019.

[39] R. D. Venkatasubramanyam and S. G. R. Why is dynamic analysis not used as extensively as static analysis: An industrial study. In *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*, SER&IPs 2014, page 24–33, New York, NY, USA, 2014. Association for Computing Machinery.

[40] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

[41] Z. Wu. Ziweiwu-github/glade-config-file, Jun 2022. https://github.com/ZiweiWu-github/GLADE-config-file.

[42] Z. Wu, E. Johnson, W. Yang, O. Bastani, D. Song, J. Peng, and T. Xie. Reinam: Reinforcement learning for input-grammar inference. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 488–498, New York, NY, USA, 2019. Association for Computing Machinery.

[43] X. Yang, Y. Chen, E. Eide, and J. Regehr. Csmith. https://embed.cs.utah.edu/csmith/.

[44] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[45] M. Zakeri. M-zakeri/iust_deep_fuzz. https://github.com/m-zakeri/iust_deep_fuzz.

[46] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Fuzzing with grammars. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2022. Retrieved 2022-01-12 14:39:50+01:00.