REINAM: Reinforcement Learning for Input-Grammar Inference

Zhengkai Wu
zw3@illinois.edu
University of Illinois at Urbana-Champaign
Illinois, USA

Evan Johnson
enjhnsn2@illinois.edu
University of Illinois at Urbana-Champaign
Illinois, USA

Wei Yang
wei.yang@utdallas.edu
University of Texas at Dallas
Texas, USA

Osbert Bastani
obastani@seas.upenn.edu
University of Pennsylvania
Pennsylvania, USA

Dawn Song
dawnsong@cs.berkeley.edu
University of California, Berkeley
California, USA

Jian Peng
jianpeng@illinois.edu
University of Illinois at Urbana-Champaign
Illinois, USA

Tao Xie
taoxie@illinois.edu
University of Illinois at Urbana-Champaign
Illinois, USA

ABSTRACT

Program input grammars (*i.e.,* grammars encoding the language of valid program inputs) facilitate a wide range of applications in software engineering such as symbolic execution and delta debugging. Grammars synthesized by existing approaches can cover only a small part of the valid input space mainly due to unanalyzable code (*e.g.,* native code) in programs and lacking high-quality and high-variety seed inputs. To address these challenges, we present REINAM, a reinforcement-learning approach for synthesizing probabilistic context-free program input grammars without any seed inputs. REINAM uses an industrial symbolic execution engine to generate an initial set of inputs for the given target program, and then uses an iterative process of grammar generalization to proactively generate additional inputs to infer grammars generalized from these initial seed inputs. To efficiently search for target generalizations in a huge search space of candidate generalization operators, REINAM includes a novel formulation of the search problem as a reinforcement learning problem. Our evaluation on 11 real-world benchmarks shows that REINAM outperforms an existing state-of-the-art approach on precision and recall of synthesized grammars, and fuzz testing based on REINAM substantially increases the coverage of the space of valid inputs. REINAM is able to synthesize a grammar covering the entire valid input space for some benchmarks without decreasing the accuracy of the grammar.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338958

CCS CONCEPTS

• **Theory of computation** → *Program analysis*; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

reinforcement learning, grammar synthesis, dynamic symbolic execution, fuzzing

ACM Reference Format:
Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: Reinforcement Learning for Input-Grammar Inference. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338958

1 INTRODUCTION

Many programs take strings of symbols as inputs. The set of such strings that a program accepts is called a language, which is represented by a program input grammar. Program input grammars facilitate understanding of the input structure and are essential for a wide range of applications such as symbolic execution [20, 28] (generally for test-input generation), reverse engineering, protocol specification [18], delta debugging [29], prevention of exploits [34, 39], and improvement of system resilience [33]. Despite the importance of program input grammars, acquiring the grammars often requires much manual effort, and these grammars are often either not specified or specified in a machine-unfriendly form (*e.g.,* text documents). For example, the full specification of the PDF format is available only in the form of a text document with over 1,300 pages [1].

For a program whose input grammar is not specified in a machine-friendly form, existing approaches have been proposed for attempting to infer the input grammar using program analysis [12, 13, 22, 23], language induction [11, 14, 16, 25, 31], and machine learning [17, 21, 32]. However, these existing approaches of grammar inference are not able to produce grammars of sufficient quality

(in terms of completeness and accuracy) for real-world software systems due to the following three main challenges.

Unanalyzable code. Existing approaches based on program analysis [12, 13, 22, 23] infer input grammars based on static-analysis information of the target program's code or from runtime program information collected via instrumentation of the program. However, these approaches cannot handle programs that cannot be instrumented (such as web services) or parts of programs that are too difficult for static program analysis to handle (such as native code or dynamic language features).

Low variety and quality of seed inputs. Existing approaches based on language induction [11, 14, 16, 25, 31] leverage language induction algorithms to synthesize input grammars given a set of seed inputs. However, the effectiveness of language induction algorithms heavily depends on the variety and quality of the seed inputs. For example, to infer an input grammar for a program that parses IP addresses, if the seed inputs contain only IPv4 addresses, then the grammar inferred by the language induction algorithms cannot capture the IPv6 format.

Lack of seed inputs. Given a large number of seed inputs, existing approaches based on machine learning [17, 32] can train machine learning models representing input grammars that can be used to generate inputs for fuzz testing. However, there are often not sufficiently many valid examples to learn from.

We aim to tackle the preceding challenges by addressing a key limitation of existing state-of-the-art approaches for grammar inference (e.g., Glade [14]). Existing approaches usually leverage active learning—*i.e.,* they use an iterative process of generalization steps, each of which generates new candidate grammars from the given seed inputs (see Section 2 for details). Such approaches discard a candidate grammar if *any* of its generated strings are rejected by the target program [14]. This design choice is common in active learning approaches, resulting in a rigid strategy of "no overgeneralization allowed" to ensure that candidate grammars in each generalization step are precise (*i.e.,* all strings generated from a synthesized grammar are covered by the ideal input grammar).

However, this design choice forgoes the opportunity to potentially expand the coverage achieved by the final synthesized grammar (*i.e.,* the overlapping scope of all the strings generated from the final synthesized grammar and all the ones generated by the ideal input grammar). For instance, a generalization operator may increase coverage at the expense of a tiny amount of overgeneralization, yet existing approaches would still reject such a generalization. Furthermore, as we illustrate in Section 3, even if a generalization operator often results in an inaccurate grammar, the composition of multiple such inaccurate generalizations may complement each other in a way that forms an accurate composite generalization.

We propose REINAM, a novel framework that uses reinforcement learning to synthesize program input grammars. In particular, REINAM improves over existing approaches by retaining the ability to accept inaccurate generalization steps while still synthesizing a final grammar that achieves high accuracy. REINAM achieves this goal using three key design choices. First, rather than representing the grammar as a deterministic context-free grammar (CFG), REINAM represents the accuracy of each production rule as a probability in a probabilistic context-free grammar (PCFG). This representation allows REINAM to quantify the quality of a candidate production rule beyond simply whether the rule overgeneralizes. For example, it enables REINAM to retain an inaccurate production rule during a single generalization step, and later decrease/eliminate the inaccuracy via a composite generalization.

Second, REINAM enhances the completeness of the final synthesized grammar by *incrementally improving* imperfect candidate grammars instead of *discarding* these grammars as done by the rigid "no overgeneralization allowed" strategy used in prior approaches [14]. In particular, REINAM incrementally adjusts the probability of candidate production rules in the PCFG model using machine learning to make the rules more accurate. A challenge is that a large dataset of seed inputs is usually needed to train a probabilistic generative model such as PCFG. Thus, REINAM uses reinforcement learning [30] to tune the PCFG. In particular, reinforcement learning uses the generative PCFG model itself to generate new training data. Then, REINAM runs the target program as a black-box oracle to check whether the generated inputs are valid, and uses this feedback to improve the PCFG model. Finally, REINAM iteratively generates more data to further tune the PCFG.

Figure 1 shows how REINAM formulates the grammar synthesis task as a reinforcement learning problem. In our formulation, an agent (the PCFG) is interacting with an environment (the target program). The agent chooses an action (the choice of productions to use to generate a program input) that causes a state transition (the portion of the program input constructed so far). Upon taking an action, the agent observes the next state. Eventually (once the program input is completely constructed), the agent receives a reward from the environment (based on whether the input is accepted or rejected). Then, REINAM uses a reinforcement learning algorithm to update the agent parameters (the PCFG probabilities).

A key challenge is that REINAM needs to adjust not only the probabilities of the PCFG, but also the structure of the PCFG (*e.g.,* by adding new candidate productions) to synthesize a more general grammar. However, traditional reinforcement learning algorithms (*e.g.,* Deep Q-learning [30]) tune only the parameters of the agent (*e.g.,* a deep neural network). Thus, in addition to using reinforcement learning to adjust the weights of the PCFG, REINAM additionally adjusts the PCFG using *generalization operators* that modify the structure of the PCFG. In particular, REINAM first applies generalization operators to construct candidate grammars, then optimizes the probabilities of the corresponding PCFG, and finally uses the PCFG probabilities to determine whether to accept productions in the candidate grammar.

Finally, REINAM uses automatic test generation algorithms to generate additional seed inputs [38, 40]. By doing so, REINAM improves generalization and alleviates the shortcomings of existing state-of-the-art approaches such as Glade [14] caused by their focus on avoiding overgeneralization.

We evaluate REINAM on 11 real-world benchmarks with manually written grammars used in real scenarios. We measure the precision and recall of the synthesized grammars, as well as the benefits of the synthesized grammars in grammar-based fuzz testing. Our evaluation results show that REINAM outperforms Glade in terms of precision, recall, and fuzz testing coverage for most of the benchmarks. In one of our benchmarks—namely, the input grammar encoding regular expressions that are accepted by the GNU Grep [7]—REINAM improves recall from 0.02 to 1.0, indicating

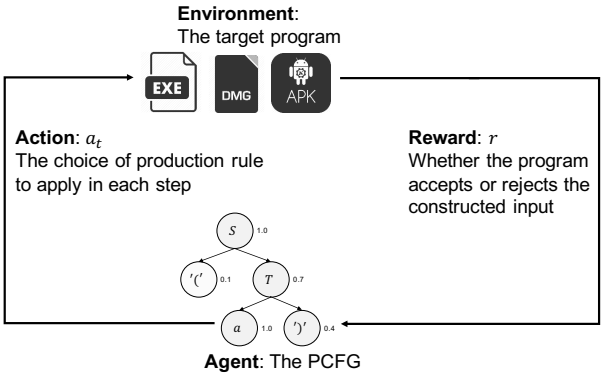

Figure 1: Formulation of grammar synthesis as a reinforcement learning problem.

that the grammar inferred by REINAM actually covers the entire program input space.[1]

In summary, this paper makes the following main contributions:

- A novel formulation of grammar synthesis as a reinforcement learning problem with a PCFG as the agent, sampled inputs as the actions, and input acceptance as the rewards.
- A corresponding learning algorithm, called REINAM, which iteratively chooses a generalization operator to apply to the current PCFG, adjusts the probabilities of the PCFG using reinforcement learning, and then retains only general and accurate production rules.
- An evaluation of on 11 real-world benchmarks for showing that REINAM effectively synthesizes program input grammars, and furthermore improves the effectiveness of fuzz testing.

2 BACKGROUND

Grammar synthesis. We first describe Glade [14], an existing state-of-the-art approach for synthesizing a program input grammar from a given set of seed inputs specified by the users for a program that includes input validation. Glade requires only black-box access to the program and uses the program as an oracle in order to determine whether a given input is valid. In particular, Glade iteratively generates new candidate grammars by applying generalization operators (from a predefined set) to the given seed inputs. Glade then checks the correctness of these candidate grammars by generating new inputs from the candidate grammars and seeing whether the new inputs are accepted by the target program. Glade consists of two steps (which they call phases). The first step learns a regular grammar by applying generalization operators such as *repetition* and *alternation* on the given seed inputs. The second step transforms the learned regular grammar into a context-free grammar by applying *merging* operators. Between the first and second steps, the *character generalization* operator is applied to generalize characters. The following includes more details:

- *Alternation:* Decompose a substring in the grammar (inside a repetition) and form an alternation. For example, "$(ab)^*$" can be generalized to $(a \mid b)^*$.
- *Repetition:* Repeat a given substring in the grammar. For example, "a" can be generalized to "$a(a)^*$".
- *Merging:* Equate two non-terminal symbols in the context-free grammar translated from the regular grammar resulted from Step 1. For example, suppose we have a CFG $S \rightarrow (\text{'}a\text{'}\ T\ \text{'}a\text{'})^*; T \rightarrow (\text{'}b\text{'} \mid \text{'}c\text{'})^*$. [2] We can merge S and T by substituting T with S, so the grammar becomes $S \rightarrow (\text{'}a\text{'}\ S\ \text{'}a\text{'})^*; S \rightarrow (\text{'}b\text{'} \mid \text{'}c\text{'})^*$.
- *Character generalization:* Allow certain terminal symbols (*e.g.,* $\text{'}a\text{'}$) to be substituted for other ones (*e.g.,* $\text{'}b\text{'}$, $\text{'}c\text{'}$, ...).

Glade performs a set of checks to avoid overgeneralization. In particular, Glade constructs a set Chk of strings such that each $\alpha \in Chk$ uses the candidate production rule added by the generalization operator in its derivation. Then, Glade executes the program on each $\alpha \in Chk$ and determines whether it is accepted or rejected. If *any* α is rejected, then Glade rejects this generalization. This mechanism is designed to enforce the "no overgeneralization allowed" strategy. However, generalization can still occur, because it could be the case that all $\alpha \in Chk$ are accepted, but there exists other inputs generated by the grammar that would be rejected by the program. Indeed, in our evaluation, we find that Glade occasionally overgeneralizes.

Probabilistic context-free grammar (PCFG). A PCFG is a CFG augmented with a probabilistic distribution. In particular, a PCFG G is a tuple $G = (M, T, R, S, P)$, where M is the set of non-terminal symbols, T is the set of terminal symbols, R is the set of production rules, S is the start symbol, and P is a set of the probability distributions over production rules. More precisely, for each non-terminal symbol A, if there are k different production rules $r_1, ..., r_k$ with A as the left-hand side, then $P_A(i)$ is the probability of choosing production rule r_i. These probabilities should satisfy $\sum_{i=1}^{k} P_A(i) = 1$ and $P_A(i) \geq 0$.

3 MOTIVATING EXAMPLE

A key shortcoming of Glade is its reliance on the "no overgeneralization allowed" condition. In this section, we give an example showing why this design choice can be problematic. In particular, we find that as a consequence of this design choice, the Grep program used in the evaluation of Glade can achieve only very low coverage of the valid input space [14].

Figure 2 shows that the grammar of the Grep program consists of many special characters (*nbchar* and *npchar*). These special characters not only form the basic building blocks for the grammar (*char*), but also serve as the special control characters in the grammar (see the production rule of *single*). Therefore, in Step 1 of Glade (*i.e.,* generalization to a regular language), the generalization process can fail. For example, suppose the seed input is "$[\wedge a]$"; then, the generalization operators in Step 1 are unable to cover the grammar "$[= (a)* =]$" (here, $\text{'}=\text{'}$ can also be $\text{'}.\text{'}$ or $\text{'}:\text{'}$ since they appear in the same production rule). The best that we can do is to apply the repetition operator, in which case the grammar is generalized to "$[\wedge(a)*]$". Next, at the intermediate step of character generalization step, we consider the generalization "$[(\wedge| =)(a)*]$", but find that

[1] For conciseness, we describe the key features of our approach in this paper and relegate implementation details to an appendix available on our project website: https://sites.google.com/site/reinamlearning/.

[2] Note that this CFG is not in normal form.

$$
\begin{array}{ll}
S \rightarrow' a' \; T \; 'a' & \\
T \rightarrow P \; Q \; T \mid P \; Q & \\
P \rightarrow \ldots & \\
Q \rightarrow \ldots &
\end{array}
\Longrightarrow
\begin{array}{ll}
S \rightarrow' a' \; T \; 'a' & \\
T \rightarrow P \; T \mid Q \; T \mid P \mid Q & \\
P \rightarrow \ldots & \\
Q \rightarrow \ldots &
\end{array}
$$

$$
\Longrightarrow
\begin{array}{l}
S \rightarrow' a' \; S \; 'a' \\
S \rightarrow P \mid Q \mid P \; S \mid Q \; S \\
P \rightarrow \ldots \\
Q \rightarrow \ldots
\end{array}
$$

Figure 3: An example of an initial grammar (top left), and two generalization steps (top right, bottom).

$$
\begin{array}{l}
genchar \rightarrow' 0' \mid 'A' \mid 'a' \\
\quad nbchar \rightarrow genchar \mid s \mid t \mid '!' \mid '"' \mid '\#' \mid '\$' \mid '\%' \mid '\&' \mid '" \\
\qquad\quad \mid ',' \mid '.' \mid '/' \mid ' :' \mid ';' \mid ' <' \mid ' =' \mid ' >' \\
\qquad\quad \mid '@' \mid '\_' \mid '\backslash' \mid ' \sim' \\
\qquad \ldots \\
\quad single \rightarrow single \; '[' \; tok \; ']' \mid single \; '[' \; ' \wedge' \; tok \; ']' \\
\qquad\quad \mid single \; '[' \; ' =' \; tok \; ' =' \; ']' \mid single \; '[' \; '.' \; tok \; '.' \; ']' \\
\qquad\quad \mid single \; '[' \; ' :' \; tok \; ' :' \; ']' \\
\quad regex \rightarrow single \mid regex \; single \\
\quad regex \rightarrow regex \; '\backslash' \; '(' \; regex \; '\backslash' \; ')'
\end{array}
$$

Figure 2: The ground-truth grammar of Grep (some rules are omitted).

this generalization is invalid. Thus, Glade is unable to insert the $' ='$ both before and after "$(a)*$" to generalize the grammar to cover "[= $(a)*$ =]".

This example demonstrates two shorcomings of Glade. First, it shows that Glade is very sensitive to the given seed inputs. For example, Glade will not cover "[= $(a)*$ =]" unless the user provides a seed input that includes an expression of the form "[= a =]". In the evaluation of Glade's grammar generation, the authors use 50 seed inputs that are randomly generated from the ideal grammar. However, in a real world scenario, the developers using Glade most likely do not know the ideal grammar—otherwise, they would not need to use Glade. Therefore, we can expect that the quality of seed inputs will be far worse than those sampled from the ideal grammar.

Additionally, the grammar of Grep queries is very coarse in the sense that special characters can be used as both the content of queries as well as control characters in the queries. This property makes Grep challenging for Glade, since the seed inputs must cover the behaviors of all the special characters. Thus, randomly generated seed inputs are not enough for Glade to synthesize the desired grammar.

Since the quality of seed inputs greatly affects the performance of Glade, we propose to use test generation tools such as Pex [38, 40] to *automatically generate* seed inputs. Pex is a white-box automated testing tool based on dynamic symbolic execution. It explores possible program execution paths to generate test inputs that cover as many parts as possible of the program. Our results show that we substantially increase both the precision and the recall on the Grep benchmark with the help of Pex.

The second shortcoming is that Glade works very hard to avoid overgeneralizing; even when overgeneralization occurs, it is due to a shortcoming in the checks used to detect overgeneralization rather than a deliberate choice. In our example, to generalize "[a.]" to a grammar that covers "[= a =]", Glade would have to perform character generalization in two places simultaneously (*i.e.,* at each of the two '=' characters). If we instead allow for overgeneralization, then we can keep the intermediate grammar "([= a(= | .)])*" after applying character generalization to the second "=". Note that this grammar can generate the input "[= a.]" being rejected by Grep, so Glade does not retain this generalization. However, if we subsequently apply another character generalization to the first $' ='$, then grammar can be transformed into "([(= | .)a(= | .)])*", which increases coverage since it now covers "[= a =]". Thus, Glade's strategy of "no overgeneralizations allowed" makes performing this pair of generalization steps impossible.

In contrast, REINAM represents the grammar as a PCFG and uses reinforcement learning to adjust the probabilities of this PCFG to improve performance.[3] Thus, REINAM can retain some amount of overgeneralization at each step, enabling it to achieve larger coverage while sacrificing only a small amount of accuracy. This ability is even more powerful if the ideal grammar is not context-free. In this case, REINAM would learn an overgeneralized grammar that can achieve high coverage, whereas Glade would frequently fail to generalize due to non-context-free constraints on valid inputs that it is unable to capture.

Finally, the third shortcoming of Glade is that the generalization operators are divided rigidly into two steps—*i.e.,* repetition and alternation in Step 1, and merging in Step 2. Glade performs only the repetition and alternation in Step 1 and transforms the resulting regular expression into a CFG in Step 2. However, the repetition and alternation are still viable and often needed in Step 2.

For example, consider Figure 3. Suppose that the left-most grammar is an intermediate state during execution of Step 2 of Glade. The middle and right grammar are constructed by first applying an alternation operator that changes the production of T form $P \; Q$ to $P \mid Q$ (left to middle), and then applying a merging operator on the symbols T and S. However, in Glade, such a generalization is not possible since repetition and alternation are not performed in Step 2. In contrast, REINAM combines the two steps of Glade—REINAM can perform any of the different kinds of generalization steps on the current CFG.

4 REINAM

At a high level, REINAM takes as input the target program for which we want to synthesize an input grammar, and then proceeds in two phases. In Phase 1, REINAM generates high-variety, high-quality seed inputs using automatic test generation (*e.g.,* the symbolic execution engine Pex [38]), and then uses an existing grammar synthesizer (*e.g.,* Glade) to synthesize an initial CFG. In Phase 2, REINAM converts the CFG from Phase 1 to a PCFG, and then uses reinforcement learning to refine this PCFG. An overview of REINAM is shown in Figure 4.

[3]Note that Glade uses a PCFG to generate new inputs only for fuzzing, not for synthesis.

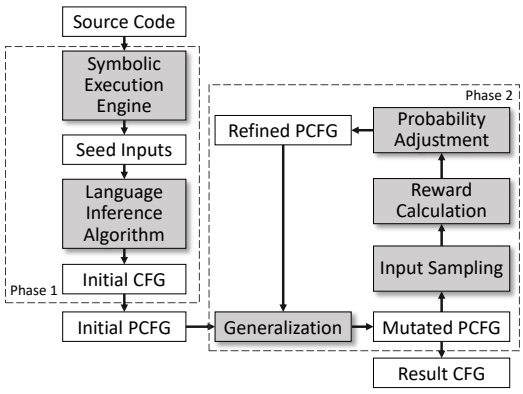

Figure 4: Workflow of REINAM.

Reinforcement learning consists of iteratively performing five steps: (i) apply generalization operators, (ii) sample strings from the PCFG, (iii) calculating the reward for each production rule, (iv) adjusting the probability distribution based on the calculated reward, and (v) removing low-probability rules. Note that the generalization operators applied to modify the current PCFG can be customized for different grammars, including operators beyond the ones described in this paper, thereby bringing flexibility to our approach.

4.1 Phase 1: Generating Seed Inputs using Pex

Phase 1 first runs a symbolic execution engine (we use Pex [38]) on the assembly code of the target program. Next, it runs a grammar synthesizer (we use Glade) using the output of the engine as the seed inputs. Pex performs path-bounded dynamic symbolic execution by repeatedly executing the program to generate path-based constraints and using an SMT (satisfiability modulo theory) solver to solve these constraints to obtain the program inputs that would lead to different execution paths. As discussed in Section 3, the quality of a set of seed inputs is primarily determined by the various "categories" of inputs that the set can cover, corresponding to inputs that achieve high code coverage. Test generation tools such as Pex are designed to generate inputs that achieve high code coverage.

4.2 Phase 2: Generalizing PCFGs via Reinforcement Learning

Initializing the PCFG. Phase 2 first converts the CFG from Phase 1 to a PCFG. For each nonterminal S, we count the number k of production rules that expand S (*i.e.,* S is on the left-hand side of the rule); then, we assign each of these rules probability $\frac{1}{k}$.

Reinforcement Learning. Reinforcement learning (RL) is an area of machine learning inspired by behavioral psychology. The basic idea is that software *agents* take *actions* in the *environment* to maximize a given metric called the *reward*. In our context of the grammar synthesis problem, these concepts are

- *Agent:* The PCFG.
- *Environment:* The target program.
- *State:* A partial derivation of the PCFG (*i.e.,* a sequence α consisting of both terminal and non-terminal symbols).
- *Action:* The choice of the production rule to apply to one non-terminal symbol in α in current state.
- *Reward:* Whether the constructed input is accepted by the target program—*i.e.,* 1 if it is accepted and 0 if it is rejected.

In other words, the agent is taking actions that choose producitons used to construct a program input. For the PCFG agent, the productions are chosen randomly according to the PCFG probabilities; we describe this process in detail below. The reward is whether the input constructed by the agent is accepted by the program.

The goal of RL is to optimize parameters of the agent so it takes actions that maximize the reward. The purpose in our context is to increase the quality (*i.e.,* precision and recall) of the grammar. However, existing RL algorithms are designed to maximize real-valued parameters [30]. In contrast, we may also need to modify the structure of the PCFG itself (*i.e.,* which productions are available). To do so, we interleave a traditional RL algorithm with the application of generalization operators to the PCFG. In particular, we iteratively perform the following: (i) apply a generalization operator, (ii) run a traditional RL algorithm—in particular, policy gradients [37]—to adjust the PCFG probabilities, and (iii) remove any production rule with probability lower than a fixed threshold from the PCFG.

Generalization operators. We use four kinds of generalization operators. First, we use the character generalization operator described in Section 2, adapted to work for PCFGs. We assign the probability of $\frac{1}{\text{\# current characters}}$ to the newly added character and reduce other probabilities proportionally.

Second, the repetition operator changes part of the grammar from "p" to "$p(p)*$". In particular, the operator picks a production rule and tries to repeat the symbols on the right-hand side—*i.e.,* for the rule $S \rightarrow P\ Q$, we would try two generalizations: (i) add the rule $S \rightarrow P\ P*\ Q$, and (ii) add the rule $S \rightarrow P\ Q\ Q*$.

Third, the alternation operator changes part of the grammar from "pq" to "$p \mid q$". In particular, the operator picks a production rule, randomly decomposes the right-hand side of the rule into two parts, and replaces the right-hand side with an alternation of these two parts—*e.g.,* for the rule $S \rightarrow P\ Q\ 'a'$, it may add new rules $S \rightarrow P1\ 'a'$ and $P1 \rightarrow P \mid Q$. The probability of the original rule $S \rightarrow P\ Q\ 'a'$ is split equally between $S \rightarrow P\ Q\ 'a'$ and $S \rightarrow P1\ 'a'$, and the probability of $P1 \rightarrow P \mid Q$ is 1.

Fourth, the merging operator merges two nonterminal symbols by substituting all usage of one symbol for another. In particular, to merge P and Q, we add rules $P \rightarrow Q$ and $Q \rightarrow P$. We assign probability $\frac{1}{\text{\# production rules of } P}$ to $P \rightarrow Q$ and reduce the other probabilities proportionally, and similarly for $Q \rightarrow P$.

These generalization operators have two advantages: (i) they are already used by Glade (and other grammar synthesis algorithms use similar operators [22, 26]), and (ii) they are simple to implement.

Constructing inputs using a PCFG. We perform the following steps to inputs from our PCFG:

- Initialize $\alpha = S$ (where S is the start symbol of the PCFG).
- While α contains non-terminal symbols, uniformly randomly choose a random non-terminal A in α, and then randomly apply a production rule to A based on the PCFG probabilities.
- Return $\alpha = \alpha_1 \ldots \alpha_k$ (now, each α_i is a terminal symbol).

After sampling an input, we execute the program on this input and record whether the program accepts or rejects the input.

Probability adjustment. Recall that in our PCFG, each probability quantifies the correctness of the corresponding production rule—*i.e.,* whether this rule exists in the ideal grammar. Unlike Glade, we allow occasional overgeneralization. In Glade, if any input generated using a new production rule is rejected by the program, then that rule is rejected. In contrast, our algorithm does not necessarily reject a rule if the rule fails a check as long as it produces at least one input that is accepted. In particular, we use reinforcement learning to automatically adjust the probability of each production rule. To do so, we track which new production rules are used to construct each input. Then, we define the following aggregate reward for each new production rule r_i:

$$\text{reward}(r_i) = \frac{\text{\# accepted inputs that use } r_i}{\text{\# inputs that use } r_i}$$

We use the policy gradient algorithm to tune the PCFG probabilities [37]. We consider different non-terminal symbols separately since the probability of a production rule is related only to those of other production rules for the same non-terminal symbol. Consider a non-terminal symbol A and the probability distribution for its productions is θ—*i.e., A* has k production rules $r_1^A, ..., r_k^A$, and the probability for rule r_i^A is $\theta(r_i^A)$. The policy gradient [37] update gives us the following adjusted probabilities θ':

$$\theta' = \theta + \eta \cdot \nabla_\theta \log \pi_\theta(s_t, a_t) \cdot v,$$

where $\pi_\theta(s_t, a_t)$ is the probability that the agent chooses action a_t in state s_t, η is a fixed learning rate, and v is the reward. In our setting, we have

$$\begin{aligned}\theta'(r_i^A) &= \theta(r_i^A) + \eta \cdot \nabla_\theta \log(\theta(r_i^A)) \cdot \text{reward}(r_i^A) \\ &= \theta(r_i^A) + \eta \cdot \frac{reward(r_i^A)}{\theta(r_i^A)} \end{aligned} \quad (1)$$

Finally, we normalize the new probability distribution $\theta'(r_i^A)$ so it sums to 1.[4] After each application of a generalization operator, we iteratively sample new inputs and run the policy gradient update until the PCFG probabilities converge. We use the convergence threshold $\frac{1}{10 \cdot (\text{\# production rules of } A)}$—*i.e.,* we terminate if no probability changes by more than this threshold.

Rule removal. Finally, after adjusting the probabilities, our algorithm removes production rules with probability lower than the convergence threshold (or equivalently, we set their probabilities to zero). This step plays the role of undoing failed generalization operators, and also avoids wasting computation time on adjusting the probabilities of incorrect production rules.

4.3 Discussion

Negative probability adjustments. Note that in the probability adjustment formula (1), the gradient $\frac{\text{reward}(r_i^A)}{\theta(r_i^A)}$ is non-negative. Thus, a potential concern is that the gradient can only ever increase $\theta(r_i^A)$. The reason why this is not an issue is that because after taking a gradient step, we normalize the updated probabilities θ' so that $\sum_i \theta'(r_i^A) = 1$. In particular, if a production rule r_i^A achieves higher reward $\text{reward}(r_i^A)$, then the gradient computed using formula (1) is larger, so the probability increases—*i.e.,* $\theta'(r_i^A) > \theta(r_i^A)$. Conversely, if r_i^A achieves lower reward, then the probability becomes decreases—*i.e.,* $\theta'(r_i^A) < \theta(r_i^A)$.

Time complexity. We can compute the time complexity of our approach by estimating the number of sampling and probability adjustment steps needed for the convergence. Suppose the threshold used by us to determine stability of the probability is ϵ (*i.e.,* if no probability changes by more than ϵ, then we terminate this round of RL and start the next generalization step), and assume that the set of sampled inputs produces the same reward each time.[5] Then, the gradient difference each step is $\eta \cdot \frac{\text{reward}(r_i^A)}{\theta(r_i^A)}$, where $\eta \cdot \text{reward}(r_i^A)$ remains constant by assumption. Thus, $\theta(r_i^A)$ converges proportionally to the ratio of accepted strings among all strings using that rule. The number of sampling and adjustment steps for convergence is approximately $O(\frac{1}{\eta \cdot \epsilon})$, which is constant.

Therefore, the (expected) time complexity of our approach is $O(n \cdot m \cdot RL(n))$, where n is the number of symbols in the grammar, $RL(n)$ is the time needed for one iteration of RL, and m is the total number of possible production rules. In particular, $O(n \cdot m)$ is an upper bound on the number of generalizations that may be tried (while generalization operators can introduce new production rules, but we apply only generalization to the initial production rules set). Also, $RL(n)$ equals the time used for sampling strings and executing the program on these input strings. The time used for probability adjustment is negligible. By our previous discussion, each generalization takes $O\left(\frac{1}{\eta \cdot \epsilon}\right)$ rounds of sampling and adjustment, so $RL(n) = O\left(\frac{(\text{total \# strings sampled}) \cdot (\text{average execution time})}{\eta \cdot \epsilon}\right)$. We discuss the choice of the number of strings sampled in our appendix available on our project website.

PCFG probabilities. We note that the PCFG probability of a production rule represents how likely a resulting string would be accepted by the program if we apply that rule, not the actual frequency with which the production rule used to construct real-world inputs. The reason is that we do not have any information about the distribution of real-world program inputs. As mentioned above, we initially set the probability of each production rule r_i^A for a fixed non-terminal symbol A to be constant. Afterwards, we are adjusting the probabilities such that they converge proportionally to the ratio of accepted strings among all strings using a given rule. We then eliminate production rules with small probability (corresponding to unreasonable overgeneralizations).

5 EMPIRICAL EVALUATION

To evaluate the effectiveness of REINAM and the contribution of each phase to the effectiveness, we conduct an empirical evaluation on 11 benchmarks. We seek to answer the following research questions:

- **RQ1**: How effective is the final grammar synthesized by REINAM in terms of precision and recall?
- **RQ2**: How effective is the final grammar synthesized by REINAM in terms of improving fuzz testing?
- **RQ3**: How do the two phases of REINAM contribute to the grammar's precision/recall and performance in fuzz testing?

[4] This normalization can formally be derived using Lagrange multipliers.

[5] This assumption may not be true in practice due to random noise; we make the assumption to simplify our analysis of convergence.

- **RQ4**: What is the execution time of REINAM?

In RQ1, we compare REINAM with Glade in terms of the precision and recall of the final synthesized grammar. We compute the two metrics using the manually written ideal grammar for the benchmarks as ground truth. In RQ2, we evaluate the capability of REINAM to learn a grammar for a fuzz testing task. We feed the grammar synthesized by REINAM to a grammar-based fuzzer to perform fuzz testing on programs. We compare our results with Glade and an industrial fuzzer. In RQ3, we compare the grammar after Phase 1 of our algorithm to the final grammar after Phase 2 of our algorithm. We compare the difference both in precision/recall and in fuzz testing coverage. In RQ4, we measure the time used by each phase of REINAM.

5.1 Benchmarks

We include the benchmarks of manually written grammars from the Glade evaluation [14]:

- A grammar used to match URLs [3]. We separate this grammar into four benchmarks based on the protocols (*i.e.,* "http", "https", "mailto", and "nntp"). We test whether REINAM can infer a complete grammar starting from seed inputs including only one of the four protocols.
- A grammar for the regular expressions accepted as input by GNU Grep [7]. This grammar is shown in Figure 2.
- A grammar for a simple Lisp parser [2], including support for quoted strings and comments.
- A grammar for an XML parser [5], including all XML constructs except that only a fixed number of tags are included (to ensure that the grammar is context-free). We use the .NET XML library and it has an alphabet of tags to choose from. Neither Glade nor REINAM are provided with the alphabet.
- A grammar for a Cascading Style Sheets (CSS) parser [4]; CSS is a language used to describe the presentation of a document written in a markup language such as HTML.

In addition to the Glade benchmarks, we include a grammar used to match IPv4 and IPv6 addresses, and two benchmarks generated from ANTLR [6], a widely used parser generator. We use these two benchmarks to evaluate the performance of REINAM on synthesizing a grammar for a program generated by an automatic parser generator. We select the following grammars (from the official website of ANTLR [6]) that have less than 100 non-terminal symbols:

- A grammar used to match CSV files [8].
- A grammar used to describe simple first-order logic (FOL) formulas [9].

The data and benchmarks of our evaluation are available on our project website.

5.2 Evaluation Setup

Precision and recall. Precision measures the probability that a randomly generated string from our synthesized lanugage L is accepted by the target program (i.e., included in the ideal language L_*), and recall measures the probability that a randomly generated string from the ideal language L_* belongs to our synthesized language L. In other words, the precision of REINAM indicates whether our synthesized language L overapproximates the ideal language L_*, and the recall indicates whether L underapproximates L_*. We calculate precision as $\frac{|E_{\text{prec}} \cap L_*|}{|E_{\text{prec}}|}$, where E_{prec} is a set of 1000 strings randomly sampled from L, and we calculate recall as $\frac{|E_{\text{rec}} \cap L|}{|E_{\text{rec}}|}$, where E_{rec} is a set of 1000 strings randomly sampled from a reference grammar used to specify L_*.

Generation of seed inputs. We use Pex's dynamic symbolic execution capabilities to generate input strings that are accepted by the parser, and then use these strings as seed inputs to Glade.

Sampling from PCFG. We sample a string matching a nonterminal A in the CFG G as follows:

- Randomly select a production $A \rightarrow A_1 \ldots A_n$ for A.
- For each $i \in \{1, \ldots, n\}$, if A_i is a nonterminal, then recursively sample a string matching A_i, and if A_i is a terminal, then return A_i.

For simplicity, we use a uniform distribution over productions when sampling strings to measure precision and recall. In contrast, our RL algorithm samples strings using the probabilities in the PCFG G. Here, we discard the probabilities since they do not necessarily capture the true distribution of inputs.

Programs used for evaluation. Because Pex requires source code to perform dynamic symbolic execution and generate program inputs, we write C# programs to parse the grammars described in Section 5.1. We use parsers in the .NET system library to parse the grammars of URLs, IP addresses, regular expressions, and XML documents. We also test against an open-source Lisp parser and an open-source CSS parser. The programs for the two ANTLR grammars are generated by using ANTLR in C# mode.

Fuzz testing. Fuzz testing (or fuzzing) is an automated software testing technique. An effective fuzzer generates "sufficiently valid" inputs and then monitors the execution of the program on these inputs. One of the purposes of using fuzz testing is to observe the behavior of the target program under various inputs, so we want the fuzzer to achieve high code coverage.

Grammar-based fuzzer. Both Glade and REINAM synthesize a CFG L that approximates the ideal language L_*. We use a grammar-based fuzzer to leverage the synthesized grammar to improve fuzzing. The fuzzer is similar to the step of input sampling described earlier. We randomly sample 1000 strings from the grammar, using a uniform distribution over productions. As before, for REINAM, we use a uniform distribution instead of using the PCFG probabilities since these probabilities have no relation to the distribution of real inputs. Then, we execute the program on the sampled inputs and use Visual Studio to measure the code coverage as the number of blocks covered. We use the code coverage achieved using the manually written ideal grammar as an upper bound. We compare our fuzzer against a state-of-the-art fuzzer that does not employ grammar-based fuzzing, Radamsa [10]. Radamsa combines random bit-flipping with domain-independent heuristics designed to test edge cases to create high quality mutation. Note that the Learn & Fuzz [21] tool is not available to us, so we cannot compare against their approach.

5.3 RQ1: Precision and Recall

As shown in Table 1, REINAM performs well across all benchmarks. The row "LI" shows the result of the grammar synthesized by solely

Table 1: Precision (P) and Recall (R).

Benchmark	http		https		nntp		mailto		ip		css		xml		grep		lisp		csv		fol	
	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
LI	0.79	0.06	0.77	0.07	0.80	0.05	0.99	0.13	1.00	0.50	0.99	1.00	0.32	1.00	0.79	0.02	0.65	1.00	1.00	0.24	0.90	0.30
LI+SE	0.96	0.50	0.96	0.64	0.97	0.49	0.97	0.42	0.91	1.00	0.99	1.00	0.56	1.00	0.98	0.40	0.91	1.00	1.00	0.40	0.88	0.45
LI+SE+RL	0.87	0.77	0.91	0.70	0.95	0.82	0.93	0.77	0.82	1.00	0.86	1.00	0.50	1.00	0.78	1.00	0.73	1.00	0.94	1.00	0.75	0.45

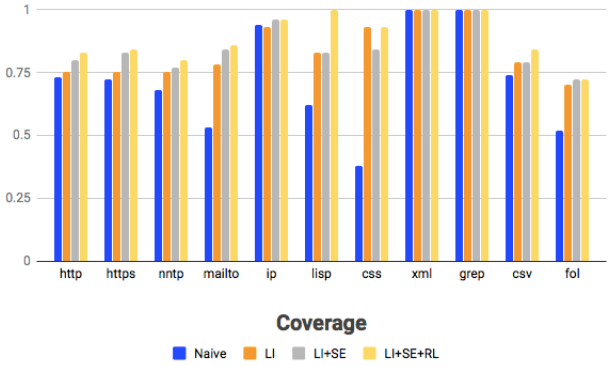

Figure 5: Code coverage achieved by different fuzzers (as percentages)

Table 2: Coverage (numbers are #basic blocks covered in code).

Benchmark	http	https	nntp	mailto	ip	lisp	css	xml	grep	csv	fol
Radamsa	1579	1558	1461	1157	2272	131	737	1349	1819	82	216
LI	1625	1623	1618	1691	2241	175	1819	1349	1819	88	348
LI+SE	1750	1796	1673	1829	2318	176	1642	1349	1819	88	353
LI+SE+RL	1784	1811	1716	1863	2325	211	1809	1349	1819	92	353
Total	2164	2164	2164	2164	2421	211	1948	1349	1819	111	414

LI (**L**anguage **I**nference algorithm): results of Glade only
LI+SE (**S**ymbolic **E**xecution engine): results of REINAM Phase 1
LI+SE+RL (**R**einforcement **L**earning): results of REINAM Phase 1+2
Radamsa: code coverage achieved by Radamsa

running Glade, while the row "LI+SE+RL" shows the result of the final grammar synthesized by REINAM. On average, the precision stays almost the same, with a less than 1% change (-0.36% on average), while the recall improves drastically by 49.2%. This rate is calculated by subtracting the number on the third row by the first row and taking the average across benchmarks. On all of our 11 benchmarks, REINAM outperforms Glade in recall without losing precision. In the three benchmarks "css", "xml", and "lisp", Glade already achieves 1.0 recall, indicating that the grammar synthesized by Glade already covers the entire input space of the three programs. For these benchmarks, REINAM cannot further improve recall, but manages to keep the perfect score as expected.

As mentioned in Section 5.1, we intentionally split the "url" grammar into four benchmarks "http", "https", "nntp", and "mailto". For these four benchmarks, we see a substantial increase in recall. Glade performs poorly, with recall lower than 0.15 on all four benchmarks, due to the limitation of Glade that the quality of the synthesized grammar highly depends on the quality of the seed inputs. Since Glade starts with input strings starting with "http" ("https", "nntp", or "mailto" in other three benchmarks) as the seed inputs, Glade is not able to explore other possible URL protocols. Meanwhile, REINAM leverages the help of Pex in Phase 1 in order to generate a set of seed inputs for achieving higher coverage. Therefore, the final synthesized grammar of REINAM for these four benchmarks has recall above 0.70 without substantial loss of precision. The "ip" benchmark is similar. Glade starts with the seed inputs in IPv4, is unable to generalize the grammar to cover the IPv6 case.

The performance on benchmarks "grep" and "csv" is similar. We have already analyzed why Glade achieves only 0.02 recall on "grep" in Section 3. We can see that REINAM gets 1.0 recall—i.e. all possible strings that can be accepted by the program are covered by the final synthesized grammar of REINAM. Given that the precision is similar (0.78 vs. 0.79), the improvement is substantial. The "csv" benchmark is similar to the "grep" benchmark in the sense that it includes many characters with special behaviors. In particular, since the grammar describes all possible CSV files, there are many special characters that can be in a data field, and several special characters serve as both separating symbols and content symbols. As discussed in Section 3, this property makes the benchmark challenging for Glade since the seed inputs must cover all of these behaviors. Therefore, REINAM outperforms Glade by improving recall from 0.24 to 1.0.

The case for the "fol" benchmark is unique. Although REINAM improves recall from 0.3 to 0.45, it decreases precision from 0.90 to 0.75. This benchmark is the only one for which REINAM cannot achieve a recall above 0.5. Intuitively, first-order logic (FOL) formulas are more complex to write. For example, to write a valid HTTP address, it only needs to start with "http://" and have a "." between the site and domain segments. However, FOL formulas are much more strictly formatted. Intuitively, the ideal language of FOL formulas is quite sparse, indicating that if we consider the entire input space of a FOL parser, it would be much smaller than the input space of a URL parser. Therefore, the generalization operators in REINAM may have difficulty generalizing to the complicated structure of the FOL grammar.

5.4 RQ2: Application in fuzzers

In the fuzz testing experiment, we can see from Table 2 and Figure 5 that REINAM greatly improves code coverage. If we compare the rows "Radamsa", "LI", and "LI+SE+RL", we find that on average, REINAM improves coverage by 18.4% compared to the Radamsa fuzzer and by 4.9% compared to Glade's grammar-based fuzzer.

In the benchmarks "xml" and "grep", all fuzzers achieve perfect code coverage. Interestingly, the grammar synthesized by Glade ("LI") achieves only 0.02 recall in Table 1, but the fuzzer using this grammar still achieves perfect code coverage. The reason is that different strings generated from the same grammar can share same or similar execution paths that would cover similar basic blocks in code. Therefore, coverage is not always correlated with recall.

Ignoring the two benchmarks with perfect coverage for all approaches, REINAM outperforms the naïve fuzzer on all benchmarks.

Table 3: Execution time breakdown of REINAM (number in seconds and percentage of total time).

Benchmark	http	https	nntp	mailto	ip	lisp	css	xml	grep	csv	fol
SE	900 (33.5%)	900 (32.3%)	900 (37.8%)	900 (35.3%)	14 (22.2%)	208 (20.5%)	900 (62.5%)	605 (32.2%)	44 (7.8%)	190 (50.5%)	307 (40.8%)
LI	624 (23.2%)	834 (29.9%)	84 (3.5%)	825 (32.3%)	14 (22.2%)	358 (35.2%)	49 (3.4%)	01 (0.0%)	167 (29.7%)	4 (1.1%)	11 (1.5%)
RL	1160 (43.2%)	1051 (37.7%)	1398 (58.7%)	827 (32.3%)	35 (55.6%)	450 (44.3%)	492 (34.1%)	1273 (67.8%)	351 (62.5%)	182 (48.4%)	435 (57.8%)
Total	2684	2785	2382	2612	63	1016	1441	1878	562	376	753

When compared to Glade's grammar-based fuzzer, REINAM performs slightly worse in the "css" benchmark, but outperforms Glade on all other benchmarks. This advantage is due to the benefit that we get from allowing overgeneralization, since REINAM produces more rejected inputs than Glade. These rejected inputs cover parts of source code that the accepted inputs cannot cover—*e.g.,* the code used to handle invalid inputs.

Another observation is that the Radamsa fuzzer performs poorly on the benchmarks of parenthesis matching pattern "lisp"and "css", and also on the benchmark "fol". Especially for the "css" benchmark, Radamsa cannot even reach 40% code coverage. The reason is that as we discussed, the ideal grammars for "lisp", "css", and "fol" are more structured than those for other benchmarks. The character-level modifications used by a non-grammar-based fuzzer cannot synthesize these structured strings, whereas the generalization operators used by REINAM can infer these kinds of structures.

5.5 RQ3: Comparison of Phases 1 & 2

From Table 1, Table 2, and Figure 5, we see that Phase 1 and Phase 2 contribute differently to the improvements over Glade in terms of precision/recall and fuzzing.

In RQ1, if we compare row "LI+SE", which shows the result of the grammar synthesized by running Glade on the Pex-generated seed inputs (REINAM Phase 1), against row "LI", which shows the result of synthesized grammar of Glade itself, we find that Phase 1 on average improves precision by 9.2% and recall by 29.1%.

In the three parenthesis matching pattern benchmarks "css", "xml", and "lisp", Glade already achieves 1.0 recall, which cannot be improved upon. However, Phase 1 does improve precision in both the "xml" and "lisp" benchmarks, and maintains the same precision in the "css" benchmark. The reason is that Pex generates a set of seed inputs with higher coverage. As described in Section 2, the first step of Glade directly synthesizes a regular grammar that only captures the seed inputs. Therefore, a set of seed inputs with higher coverage would result in an initial regular grammar with higher coverage, so fewer further generalizations are needed to infer the ideal grammar. Since imprecision happens during generalization, fewer generalizations result in higher precision. We observe similar behaviors for other benchmarks, especially when the grammar synthesized by Glade has low precision.

However, for the three benchmarks "fol", "ip", and "mailto", we find that the grammar synthesized by Glade already achieves perfect or near perfect precision. In these cases, recall improves substantially after Phase 1; however, precision after Phase 1 is reduced. This reduction is a sacrifice for achieving better recall. The reason is that a set of seed inputs with higher coverage brings more opportunities for generalization, but also causes the check mechanism in Glade to fail more frequently, resulting in more uncaught overgeneralization, which thereby reduces precision.

Next, in RQ1, if we compare row "LI+SE", which shows the result of the grammar synthesized by running Glade on the Pex-generated seed inputs (REINAM Phase 1), and row "LI+SE+RL", which shows the result of the final grammar synthesized by REINAM, we find that on average, the results after Phase 2 improve recall by 20.1% but worsen precision by 9.5%.

In particular, we observe that precision decreases for all benchmarks compared to the results of Phase 1. This finding is expected since the reinforcement learning algorithm in Phase 2 allows overgeneralization to further generalize the grammar. Compared to the "no overgeneralization allowed" strategy used in Glade, our allowance of overgeneralization leads to reduced precision.

Furthermore, we observe that in the benchmarks from the URL grammar (*i.e.,* "http", "https", "nntp", and "mailto"), and the "grep" and "csv" benchmarks, after Phase 1, the recall is still low. The highest is "https" with a recall of 0.64; all others are less than 0.50. The reinforcement learning in Phase 2 further generalizes the grammar to achieve higher coverage. Table 1 shows that among these six benchmarks, recall improves by an average of 35.1% over Phase 1 alone. In addition, "grep" and "csv" actually achieve 1.0 recall, indicating that the final grammar synthesized by REINAM can perfectly cover the entire program input space. This result suggests that our reinforcement learning approach is effective especially in cases when the input space is large but existing approaches can synthesize a grammar that covers only part of the input space.

In the fuzz testing task, the average improvement in coverage achieved by the grammar-based fuzzer using REINAM's grammar (row "LI+SE+RL") to the coverage achieved by the grammar-based fuzzer using the Phase 1 grammar (row "LI+SE") is 3.2%. Similarly, the average improvement of row "LI+SE" to row "LI" is 1.7%. From the data, we can see that Phase 2 contributes more to the improvement in coverage. This comparison shows the benefits of keeping some overgeneralized rules in our RL algorithm. We see that for benchmarks such as "css", Phase 2 does not increase recall, which remains at 1.0. However, the final synthesized grammar improves coverage by 8.6% (1809 vs. 1642) compared to Phase 1. Thus, while Phase 2 does not improve the coverage of the ideal language, it improves the actual code coverage acheived using fuzz testing. This result further demonstrates that our reinforcement learning approach is effective not only for exploring the input space of all valid program inputs, but also for generating invalid program inputs that cover execution paths that cannot be covered by valid inputs.

5.6 RQ4: Execution time

We can see from Table 3 that the RL phase (Phase 2) takes an average of 49.2% of the total execution time. Pex takes an average of 34.2% of the time in Phase 1. The execution time of Pex can be tuned by setting the timeout parameters. The timeout in the evaluation is set to 900 seconds; Pex times out in five benchmarks.

Based on the discussion in Section 4.3, the execution time of the RL phase is primarily dependent on the size of the grammar and the execution time of the program. The time overhead of the RL phase is unsatisfying; however, the bottleneck in Phase 2 is waiting for the results of executing the program on sampled strings. This step can be parallelized by executing the program on k sampled strings simultaneously. Currently, we execute the program on 1500 sampled strings in parallel on 4 threads. The results can be improved by parallelizing the program execution across more threads.

6 THREATS TO VALIDITY

Threats to external validity. Our evaluation uses manually written ideal grammars to measure precision and recall. Additionally, the grammars in evaluation benchmarks may not be complicated enough to reflect real-world scenarios. We attempt to combat this threat by selecting complex grammars such as the CSS grammar.

Threats to internal validity. We randomly sample strings from the ideal grammars for Glade to match with the increased number of seed inputs (generated by Pex) included in REINAM. The randomly generated seed inputs for Glade may not be sufficiently representative, and may affect Glade's effectiveness. The technique by which we sample from the PCFG is simpler and is potentially more biased compared to more sophisticated techniques. Also, a different learning rate η may affect the evaluation results. Nevertheless, we synthesize grammars using different η, and find that the results are quite close—*i.e.,* less than 0.5% difference in precision and recall and no difference in coverage.

7 RELATED WORK

Our approach draws on both existing work in grammar inference as well as techniques used in machine learning and automated test generation.

Inferring input grammars. Höschele et al. [22, 24] propose the AUTOGRAM approach based on dynamic taint tracing to extract syntactic entities of a given seed input. By tracing the data flow of particular characters of the seed input in the parsing method of the target program, AUTOGRAM decomposes formats into meaningful fields. However, since AUTOGRAM traces only paths taken by a given seed input, it requires that the given set of seed inputs capture all meaningful features of the input grammar in order to infer a complete grammar.

Coverage-guided fuzzing. In recent years, coverage-guided fuzzing has achieved substantial success with tools such as American Fuzzy Lop (AFL)[6] and Fairfuzz [26]. However, AFL does not support fuzzing .NET executables, and does not have an appropriate replacement that we can compare to.

Many fuzzers fail to explore paths that involve a difficult check such as string-equality comparisons. Recent advances in grey-box fuzzing have utilized lightweight program analysis to mitigate this problem. Steelix [27] tracks progress in string-comparison checks to incrementally discover inputs that can bypass these checks. Angora [19] solves path constraints via a searching algorithm based on gradient descent. Other tools, such as Driller [36], use more heavyweight program analysis to bypass these checks by symbolically executing an intermediate representation and solving constraints to bypass these checks. REINAM is able to discover the "constants" in grammar to bypass these checks in the grammar-inference process. As we can see in the inference of the URL protocol, the discovery of protocol names such as "mailto" and "https" is unlikely to be discovered using grey-box approaches.

Machine learning for fuzzing. Godefroid et al. [21] use a recurrent neural network to learn an input model and generate inputs for fuzzing with the Learn & Fuzz algorithm. They have recently formalized fuzzing as a reinforcement learning problem [17]. Their work does not produce an explicit grammar, but instead uses a generative deep neural network to serve as the grammar.

PCFG inference from examples. The problem of inferring a probabilistic context free grammar (PCFG) from a set of examples has been studied extensively for the purpose of natural language processing. Belz [15] extends standard split/merge grammar inference techniques to optimize grammars from examples, but requires a large corpus of annotated examples, which are not viable for inferring program input grammars. Scicluna and Higuera [35] propose an unsupervised approach to grammar inference without annotated examples. While they show that this approach works for small samples with respect to NLP standards (the polynomial number of examples with respect to the number of productions in ideal grammars), this approach is still not viable for inferring program input grammars. REINAM addresses these issues by expanding seed inputs using Pex before synthesizing the grammar.

8 CONCLUSION

We have presented REINAM, a reinforcement learning approach for synthesizing a PCFG that encodes the language of valid program inputs. To address the challenge of lacking high-variety and high-quality seed inputs faced by the existing approaches, REINAM includes an industrial symbolic execution engine, Pex [38], to generate initial seed inputs for the given target program, and includes a grammar-generalization loop to proactively generate additional inputs during grammar inference. In the grammar-generalization loop, instead of eliminating production rules in a candidate grammar that may not be accurate initially (as done by Glade [14], an existing state-of-the-art approach), REINAM keeps and evolves inaccurate grammars, enabling it to infer ground-truth grammars whose inference requires *composite* generalizations from the initial seed inputs. To efficiently search for such composite generalizations in a huge search space of candidate generalization operators, REINAM includes a novel formulation of the search problem as a reinforcement learning problem. Our evaluation results show that REINAM outperforms Glade on both precision and recall of the synthesized grammars, and fuzz testing based on REINAM substantially increases the coverage of the space of valid inputs. REINAM is often able to synthesize a grammar covering the whole valid input space without decreasing the precision of the grammar.

9 ACKNOWLEDGMENTS

This material is in part based upon work supported by the National Science Foundation under Grant No. CNS-1513939, CNS-1564274, CCF-1816615 and TWC-1409915. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[6] http://lcamtuf.coredump.cx/afl/

REFERENCES

[1] 2009. PDF Reference. https://www.adobe.com/devnet/pdf/pdf_reference.html
[2] 2010. Norvig Lisp. http://norvig.com/lispy.html
[3] 2010. What is a good regular expression to match a URL? https://stackoverflow.com/questions/3809401/what-is-a-good-regular-expression-to-match-a-url
[4] 2015. Grammar of CSS. https://www.w3.org/TR/CSS21/grammar.html
[5] 2015. XML standard. https://www.w3.org/standards/xml/core
[6] 2018. ANTLR. https://www.antlr3.org/
[7] 2018. GNU Grep. https://www.gnu.org/software/grep/
[8] 2019. CSV grammar. http://www.harward.us/~nharward/antlr/csv.g
[9] 2019. First Order Logic grammar. https://www.antlr3.org/grammar/1336156363937/FOL.g
[10] 2019. Radamsa. https://gitlab.com/akihe/radamsa
[11] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106.
[12] Domagoj Babić, Matko Botinčan, and Dawn Song. 2012. *Symbolic grey-box learning of input-output relations.* Technical Report UCB/EECS-2012-59. EECS Department, University of California, Berkeley.
[13] Domagoj Babić, Daniel Reynaud, and Dawn Song. 2011. Malware analysis with tree automata inference. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 116–131.
[14] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 95–110.
[15] Anja Belz. 2002. PCFG learning by nonterminal partition search. In *Proceedings of the International Colloquium on Grammatical Inference*. Springer, 14–27.
[16] Matko Botinčan and Domagoj Babić. 2013. Sigma*: Symbolic learning of input-output specifications. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 443–456.
[17] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep reinforcement fuzzing. *CoRR* abs/1801.04589 (2018). arXiv:1801.04589 http://arxiv.org/abs/1801.04589
[18] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security*. ACM, 317–329.
[19] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *Proceedings of 2018 IEEE Symposium on Security and Privacy*. IEEE, 711–725.
[20] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 206–215.
[21] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 50–59.
[22] Matthias Höschele, Alexander Kampmann, and Andreas Zeller. 2017. Active learning of input grammars. *arXiv preprint arXiv:1708.08731* (2017).
[23] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 720–725.
[24] Matthias Höschele and Andreas Zeller. 2017. Mining input grammars with AUTOGRAM. In *Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering Companion*. 31–34.
[25] Hiroki Ishizaka. 1990. Polynomial time learnability of simple deterministic languages. *Machine Learning* 5, 2 (1990), 151–164.
[26] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 475–485.
[27] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 627–637.
[28] Rupak Majumdar and Ru-Gang Xu. 2007. Directed test generation using symbolic grammars. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 134–143.
[29] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*. ACM, 142–151.
[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
[31] José Oncina and Pedro Garcia. 1992. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition*. World Scientific, 99–108.
[32] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (Oct. 2018), 25 pages.
[33] Martin Rinard. 2003. Acceptability-oriented computing. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 221–239.
[34] Martin C. Rinard. 2007. Living in the comfort zone. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. ACM, 611–622.
[35] James Scicluna and Colin De La Higuera. 2014. PCFG induction for unsupervised parsing and language modelling. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. 1353–1362.
[36] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, Vol. 16. 1–16.
[37] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of Advances in Neural Information Processing Systems*. 1057–1063.
[38] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–White box test generation for .NET. In *Proceedings of International Conference on Tests and Proofs*. Springer, 134–153.
[39] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. 2004. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 193–204.
[40] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of 2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 359–368.