PySE: Automatic Worst-Case Test Generation by Reinforcement Learning

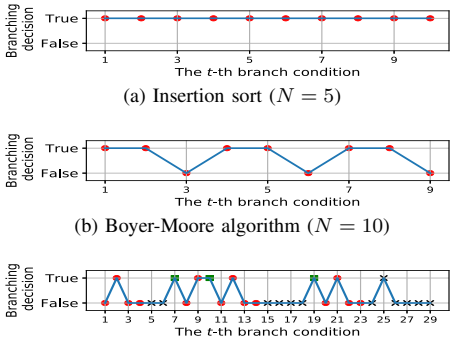
Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN
{kooj, cgusthin, milind, sbagchi}@purdue.edu

***Abstract*—Stress testing is an important task in software testing, which examines the behavior of a program under a heavy load. Symbolic execution is a useful tool to find out the worst-case input values for the stress testing. However, symbolic execution does not scale to a large program, since the number of paths to search grows exponentially with an input size. So far, such a scalability issue has been mostly managed by pruning out unpromising paths in the middle of searching based on heuristics, but this kind of work easily eliminates the true worst case as well, providing sub-optimal one only. Another way to achieve scalability is to learn a branching policy of worst-case complexity from small scale tests and apply it to a large scale. However, use cases of such a method are restricted to programs whose worst-case branching policy has a simple pattern. To address such limitations, we propose PySE that uses symbolic execution to collect the behaviors of a given branching policy, and updates the policy using a reinforcement learning approach through multiple executions. PySE's branching policy keeps evolving in a way that the length of an execution path increases in the long term, and ultimately reaches the worst-case complexity. PySE can also learn the worst-case branching policy of a complex or irregular pattern, using an artificial neural network in a fully automatic way. Experiment results demonstrate that PySE can effectively find a path of worst-case complexity for various Python benchmark programs and scales.**

***Index Terms*—Machine learning, Q-learning, Symbolic execution, Worst-case complexity, Stress testing**

I. Introduction

Stress testing is a software testing activity beyond normal operational capacity, and investigates the behavior of a program when subjected to heavy loads [1]–[4]. The goal of such tests is to observe potential functional correctness bugs (*e.g.*, deadlocks and buffer overflows) or violations in quality-related requirement (*e.g.*, latency deadline) that may not manifest under normal circumstances. Symbolic execution [5], [6] is the most widely used means to systematically construct test inputs for the stress testing. Symbolic execution runs a program using symbolic variables as inputs, instead of concrete values. When symbolic execution meets a branch condition, it can choose a specific direction of the branch as given by a branching policy. Thus, by carefully designing the branching policy, we can explore all the possible execution paths, including the ones of worst-case complexity, which are the targets of stress testing. On each path that is executed, symbolic execution collects a set of symbolic conditions, called a *path condition*. Then, it invokes a constraint solver, such as OpenSMT [7] or Z3 [8] that generates concrete test input values, which satisfy such a path condition and thus execute a program through a particular path.



(a) Insertion sort ($N = 5$)

(b) Boyer-Moore algorithm ($N = 10$)

(c) Dijkstra with a min-priority queue ($N = 12$)

Fig. 1: A sequence of branching decisions that lead to an execution path of the worst-case complexity. Here, N is the number of symbolic variables as inputs. Different markers (●,×, and ■) represent different branch conditions, meaning that there is one kind of a branch condition in (a) and (b), and there are three in (c). Note that in (c), branching decisions change irregularly and thus it is difficult to describe in a branching policy without hard-coding.

A limitation of finding the worst-case complexity by symbolic execution is that exhaustive search through all execution paths does not scale to large programs. This is because of the so-called path explosion problem, meaning that on every iteration of a loop, symbolic execution creates one new branch point, and thus the number of possible paths grows exponentially with an increase in the program input size (which typically iterates a loop more times). To deal with the path explosion problem, many efforts have been made to prune unrealizable or redundant paths based on heuristics, thereby reducing the search space of execution paths [9]–[12]. For example, Zhang *et al.* [11] proposed incremental exhaustive search that iteratively deepens search depth, pruning away similar paths at each depth among all paths created by symbolic execution. However, the main goal of the pruning-based methods is to achieve high code coverage in a feasible time bound, and they often do not focus on or fail to find the path of the true worst-case complexity, which may also get pruned out during the searching process.

Another category of solutions to avoid the path explosion problem is represented by WISE [13] and its recent improve-

978-1-7281-1736-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ICST.2019.00023

ment, SPF-WCA [14]. They first learn a branching policy that results in a path of the worst-case complexity for small input sizes by using exhaustive search, and then they apply the learned branching policy to perform a guided search for a large input size. Although proven effective in many cases, their usefulness is, however, restricted to programs whose branching policy is simple in the worst-case complexity, like "always take a `True` branch" (see Figure 1a), or "alternate between `True` and `False` decisions at a branch point in fixed patterns" (see Figure 1b). Such cases are by no means comprehensive, leaving many exceptions. For example, a Dijkstra algorithm implemented using a min-priority queue [15], which is common practice, shows an irregular pattern as in Figure 1c, and thus it is impossible to generalize a branching policy from small scales to a large scale.

In addition, projecting the small-scale behavior into a large scale will fail if there exists discontinuity in behavior over scales due to parts of the code that are activated in a scale dependent manner [16], [17]. Such cases abound in programs. For example, by a simple `if`-condition, such as `(N > N0)`, where `N` is an input size variable and `N0` is a constant, its block statement is only executed when the input size is larger than a certain threshold, and therefore, small scale tests cannot learn such behavior. Such coding pattern exists, for instance, in data communication where bulk transfer activates some specific code path.

Our solution approach: In this paper, we address the aforementioned limitations of existing symbolic execution schemes exploring the worst-case complexity, by proposing PySE[1], which is an algorithm to incrementally update the branch policy to drive the program execution toward the worst case. PySE uses symbolic execution to collect behavioral information of a given branching policy, and updates the policy through multiple iterations of the program based on Q-learning, a model-free reinforcement learning technique [18]. By iterating symbolic execution and policy update, PySE's branching policy keeps evolving in a way that the length of an execution path continues to increase over multiple executions of the program, in a long-term trend, and eventually reaches the longest possible path, meaning the worst-case complexity. PySE can deal with other definitions of the worst-case complexity, such as maximum memory utilization. We detail these other possibilities in Section VI.

Our contributions in PySE can be summarized as follows:

1) PySE is fundamentally more efficient than exhaustive searching for the worst-case path because it limits the scope of search to mild variations of what has been found to be the worst-case path so far. This feature, in effect, prunes out most unnecessary paths, and allows PySE to work in a large scale program, where exhaustive search is not feasible due to the path explosion problem.
2) Unlike WISE and SPF-WCA, PySE learns a branching policy at a targeted large scale directly, thus allowing it to handle scale-dependent program behaviors. Most importantly, PySE can learn a complex or irregular pattern of branching decisions as well as simple patterns, thanks to an expressive artificial neural network that defines its branching policy.

We have implemented PySE using Python (v2.7) on top of the Z3 constraint solver [8] and Theano machine learning library [19].[2] PySE can analyze any Python program whose inputs can be specified by symbolic variables, without the need to modify their source codes (such as adding instrumentation code to generate logs). Experimental results demonstrate that PySE can effectively deal with various Python programs and scales, which exhaustive search cannot handle because of the path explosion, and WISE-like algorithms are not efficient due to a complex or irregular branching pattern.

II. Related work

In a large body of work, the path explosion problem of symbolic execution has been tackled by pruning away redundant paths in the middle of the searching process [9]–[12]. For example, Cadar *et al.* [12] prune out paths by bounding the number of iterations in a loop, and Zhang *et al.* [11] iteratively increase exploration depth, discarding similar paths at each depth. However, the main focus of these methods is designing a heuristic that can achieve high code coverage, rather than the worst-case complexity.

WISE [13] performs exhaustive search using symbolic execution on small scales, and learns a fixed branching decision at each branch point that leads to a path of the worst-case complexity. Then, the learned branching policy is applied for guided search in large-scale tests. This work assumes that the worst-case branching decision is fixed at every branch point regardless of scales. However, this assumption does not hold for many real-life programs, like the merge sort example in [13], where the worst case is alternation between `True` and `False` at one branch point. To resolve such an issue, WISE's improved successor, SPF-WCA [14] considers the history, which is the previous branching conditions and branching decisions there. By this, SPF-WCA can dynamically change the worst-case-leading branching decision at a branch point, depending on the history. For example, in the merge sort, it can know that `True` must go after `False`, and `False` must go after `True`. However, it usually requires domain expertise to determine the length of the history necessary to test a program. In addition, it still leaves many programs uncovered. For example, a Dijkstra algorithm implemented using a min-priority queue [15] shows an irregular pattern at the worst case, as in Figure 1c. Thus, no matter how long the history is, SPF-WCA cannot fix the branching decisions at all branch points. PySE is differentiated from this kind of work in that PySE can learn a complex or irregular branching pattern in a fully automatic manner. In addition, PySE works directly on the target scale, and thus it can handle branching behaviors that are scale-dependent.

[1]The term PySE derives from **Py**thon and **S**ymbolic **E**xecution, and is pronounced like the end of "spice".

[2]Source code of PySE is available at https://bitbucket.org/helix979/pyse.

Fuzzing is also a practical approach to testing large-scale programs in which a program is bombarded with random inputs. While FairFuzz [20] focuses on code coverage, recent studies such as SlowFuzz [21] and PerfFuzz [22] have shown that fuzzing techniques can be combined with genetic algorithms to find the pathological input. Such methods adopt guided search using a genetic algorithm to find inputs that maximize the length of an execution path. They generate a set of random inputs and keep mutating the set to achieve longer execution paths. Genetic algorithm usually works well with small scales in practice. However, when the scale increases, it is well known that the genetic algorithm can be stuck with local optimum points or converge towards arbitrary points [23], [24].

Loop summarization is a technique in symbolic execution that can also be used to estimate the worst-case complexity [25]–[27]. In such a technique, the effect of a loop is modeled in terms of the number of times a loop executes, and this is reflected on the symbolic values at the exit of the loop. Thus, this kind of work can generate input values for a given iteration of a loop. However, all loops of a program should be modeled manually one by one via static analysis. Therefore, it is hard to automate such a modeling process. Another related work is worst-case execution time (WCET) analysis [28], [29]. The major goal for this work is understanding the worst-case timing behavior of a program. This problem is commonly tackled by modeling the upper bound of execution time of a loop with some level of approximation, and thus does not aim to generate concrete test input values for exercising the worst-case paths.

Fuzzing to generate vulnerabilities or WCET execution behavior can be more directed when domain knowledge is available, such as through a domain specific language [30], as shown by [31]. The need to invoke a constraint solver is another obstacle that can make it difficult to analyze the worst-case execution path at a large scale, since it may take significant time with a long path. XSTRESSOR [32] resolves such an issue by synthesizing the execution path without using a constraint solver, based on small-scale behaviors.

III. Overview

In this section, we give a high-level overview of PySE. The main job of PySE is to find out a **branching policy** $\pi(s_t)$ for a given **state** s_t at the t-th branch condition that it encounters while a program is being symbolically executed. The branching policy $\pi(s_t)$ determines a branching decision $a_t = \pi(s_t) \in \{\texttt{True}, \texttt{False}\}$, which we also call **action**. The state s_t mainly consists of the current branch condition, previous L branch conditions, and actions taken there, as indicated in the box B1 in Figure 2 that defines the state at the branch condition C1 when $L = 2$. We call L the **history length**. Refer to Section IV-A for a formal definition of a state.

We make the branching policy $\pi(s_t)$ continue evolving in such a way that the **length of an execution path**, which we define as the number of branch conditions within a path, increases in a long-term trend, and eventually reaches the worst-case complexity. To this end, PySE iterates the following two steps:

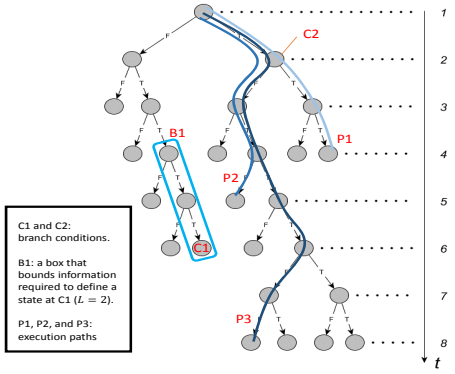


Fig. 2: An example of a computation tree where a circle denotes a branch condition at the t-th level that a program reaches during runtime.

Step 1: (Symbolic Execution) Execute a program with symbolic variables through a path decided by the branching policy $\pi(s_t)$, and collect resulting behavioral information such as branch conditions the program visits, actions taken at the branch conditions, and feasibilities of the actions.[3]

Step 2: (Policy Update) Using the behavioral information, update the branching policy $\pi(s_t)$ in a way that an undesirable action that caused a program to terminate quickly can be avoided in the future.

In other words, PySE is alternating between symbolic execution driven by the branching policy $\pi(s_t)$ in Step 1, and the learning of a good branching policy $\pi(s_t)$ in Step 2. As we will see later in Section IV-B, updating the branching policy $\pi(s_t)$ is done by Q-learning [18].

For a state s_t, we design the branching policy $\pi(s_t)$ as:

$$\pi(s_t) = \arg\max_{a_t} Q(s_t, a_t), \tag{1}$$

where $Q(s_t, a_t)$ is made from an artificial neural network (ANN), whose inputs are s_t and whose output layer produces two values, $Q(s_t, \texttt{True})$ and $Q(s_t, \texttt{False})$. That is, $\pi(s_t) = \texttt{True}$ if $Q(s_t, \texttt{True}) > Q(s_t, \texttt{False})$ and $\pi(s_t) = \texttt{False}$ otherwise. The detail of the ANN is explained in Section IV-C.

Note in (1) that $\pi(s)$ is always defined for any s, since it is described by an ANN that results in $Q(s, \texttt{True})$ and $Q(s, \texttt{False})$ for any input s. By virtue of this feature, PySE can make a branch decision even for a state that has not been observed yet. When an unobserved state comes in, ANN's internal structure will reproduce the action taken at one of the already-observed states that is recognized most similar to the input state, where similarity is internally recognized by the ANN and we do not have to explicitly define any measure of similarity.

[3]This behavioral information is defined as an experience in Section IV-D.

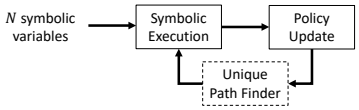


Fig. 3: Workflow of PySE that starts with specifying symbolic variables. In a basic mode, PySE works with the symbolic execution and policy update in solid boxes only. The dashed box represents the means to expedite learning of a good branching policy.

It should also be noted that the branching policy $\pi(s_t)$ is deterministic: $\pi(s_t)$ is always the same unless the function $Q(s_t, a_t)$ changes. As a result, every time we symbolically run a program exploiting $\pi(s_t)$, we will end up with an identical execution path. In order to explore a new path, PySE, thus, adds some randomness in Step 1 such that at every branch condition, PySE takes random action with small probability ϵ (typically ≤ 0.1), instead of the one decided by the branching policy. On average, the branching policy is exploited for a proportion $1 - \epsilon$ of executions, and a branching decision is selected at random for a proportion ϵ. Such exploration by random action with probability ϵ is called the ϵ-greedy strategy in reinforcement learning literature [18].

To get a better understanding of the ϵ-greedy strategy in our context, consider an example, where PySE starts with the branching policy $\pi(s_t)$ that leads to the path P1 in Figure 2. In Step 1, random action may not occur at all, since the value of ϵ is small. In such a case, the execution path is the same as P1, and PySE gathers no new information. However, if random action is chosen to occur at the branch condition C2 in Figure 2, for example, the resulting execution path will be different from P1, like P2 in Figure 2. If the new action taken at C2 results in a longer execution path, the branching policy $\pi(s_t)$ is updated in Step 2 in a manner that it is more likely to take that action again in the future. Otherwise, $\pi(s_t)$ is updated in the opposite way. Then, the updated $\pi(s_t)$ is likely to be better in Step 1 of the next iteration, resulting in a longer path like P3 in Figure 2.

As we briefly mentioned above, a naïve ϵ-greedy strategy often ends up with the path that we have already observed, thereby wasting time. In Section IV-E, we explain how to resolve this issue and expedite the learning of a good branching policy.

IV. Design Detail

Figure 3 shows the whole workflow of PySE. It takes N symbolic variables as inputs, and iterates Symbolic Execution and Policy Update, corresponding to Step 1 and Step 2 mentioned in Section III. PySE may work with these two core blocks only. We refer to such a case as a **basic mode**. We first explain the basic mode in Sections IV-A through IV-D and discuss reasons why it may be slow in learning a good branching policy. The remaining block, Unique Path Finder is an addition to expedite the learning process of the basic mode, which complete the final look of PySE. This addition is described in Section IV-E.

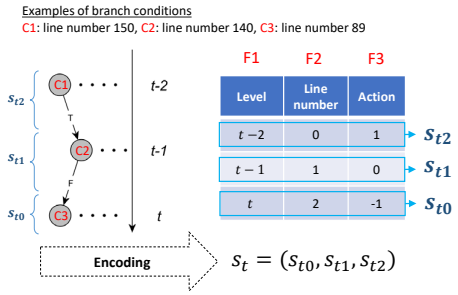


Fig. 4: Encoding of a state when $L = 2$. In the table, the field F2 encodes each line number of a branch condition into a unique integer. The F3 tells us what action was taken at a branch condition with 0 meaning `False` and 1 meaning `True`. Note that this state representation is made before PySE takes action at the t-th branch condition, so the action field in s_{t0} is set to -1 meaning the action is yet to decide. Each row of the table is the vector encoding s_{tl} for $l = 0, 1, 2$.

A. State representation

In a formal form, a state s_t at the t-th branch condition is defined as $s_t = (s_{t0}, s_{t1}, \ldots, s_{tL})$, where s_{tl} is an integer vector encoding the $(t-l)$-th branch condition and the action taken there. Figure 4 illustrates how to encode the integer vectors for a state. The content of s_{tl} includes the value of $(t-l)$ (field F1 in Figure 4), information about the line number of a program that a branch condition belongs to (field F2), and what action is taken at a branch condition (field F3). When $t \leq l$ (nonexistent cases), we set s_{tl} as an all-zero vector. In a word, the state s_t summarizes, in a machine-understandable form, representative features of an execution path from the t-th branch condition to the $(t - L)$ branch condition and actions taken in between. Note that the representation of a state s_t is scale-independent (*i.e.*, the format of s_t is the same for any N).

B. How to update the branching policy

Symbolic execution takes action a_t at a given state s_t and observes its consequence, which is whether the execution path is still feasible, *i.e.*, whether there exist any concrete input values that satisfy what is called a **path condition**, which are all the branch conditions and actions taken so far. Feasibility of the path condition can be checked by using a constraint solver [7], [8]. Depending on the feasibility, the consequence of the action a_t at the state s_t is scored by a **reward** r_t, which is defined as:

$$r_t = \begin{cases} 1 & \text{if feasible,} \\ P & \text{if infeasible,} \end{cases} \tag{2}$$

where P is called a penalty, which can be any value smaller than 1, the reward for feasible cases. We chose $P = -20$ so that the infeasible decision is more distinguishable from the feasible one. If P differs too much from 1, it can also delay the convergence of Q-network. We found by experiments that $P = -20$ is a reasonable choice that fits to our purpose. Symbolic execution stops immediately when it encounters an

infeasible path condition. Note that we are rewarded by 1 whenever the length of a feasible execution path increases by 1, and penalized by P if any action leads to an infeasible execution path. The main goal of PySE is to update the branching policy $\pi(s_t)$ so that $\pi(s_t)$ can converge to the optimal branching policy $\pi^*(s_t)$ that maximizes the expected sum of future rewards, $\mathbb{E}(\sum_{k=t}^{T} r_k|s_t)$ with T denoting the last branch condition before a program terminates normally or falls in an infeasible path condition, and thus maximizes the length of a feasible execution path. The optimal branching policy $\pi^*(s_t)$ can be of any rule, such as "always `True`" or "alternating irregularly between `True` and `False`" over t, as in Figure 1.

Now define the optimal action-value function $Q^*(s_t, a_t)$ as the maximum expected sum of future rewards, after taking action a_t at a state s_t. That is, $Q^*(s_t, a_t) = \max_\pi \mathbb{E}\left(\sum_t^T r_k \middle| s_t, a_t\right)$. Then, the optimal branching policy $\pi^*(s_t)$ can be expressed as $\pi^*(s_t) = \arg\max_{a_t} Q^*(s_t, a_t)$.

Note that $Q^*(s_t, a_t)$ can be achieved by taking action a_t at the state s_t, then continuing by choosing subsequent actions optimally. Hence, $Q^*(s_t, a_t)$ can be re-written recursively as:

$$Q^*(s_t, a_t) = \mathbb{E}\left(r_t + \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\right), \quad (3)$$

with $\max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) = 0$, by definition, when s_{t+1} is the state of program termination. In (3), since we do not know the value of $\left(r_t + \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\right)$ and its distribution for all t beforehand, direct computation of $Q^*(s_t, a_t)$ is not possible. Therefore, we will try to learn it by a sample mean $Q(s_t, a_t)$ in the following way:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha\left(r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})\right), \quad (4)$$

where α is called a learning rate. More precisely, $Q(s_t, a_t)$ is updated in a weighted mean where a new sample is added by a weight α. By the law of large numbers, $Q(s_t, a_t)$ can converge to $Q^*(s_t, a_t)$ after iterations for a sufficiently small value of α [18], [33]. Such an update for learning $Q^*(s_t, a_t)$ without knowing the underlying probability distribution model is referred to as Q-learning in the reinforcement learning literature. Using this function $Q(s_t, a_t)$, our branching policy $\pi(s_t)$ is defined as in (1). When $Q(s_t, a_t)$ converges to the optimal action-value function $Q^*(s_t, a_t)$, the branching policy $\pi(s_t)$ also converges to the optimal branching policy $\pi^*(s_t)$.

In the meantime, if we re-write (4) as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Delta Q(s_t, a_t), \quad (5)$$

where

$$\Delta Q(s_t, a_t) = r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t), \quad (6)$$

then we can see that $Q(s_t, a_t)$ converges when the magnitude of $\Delta Q(s_t, a_t)$ becomes zero. Thus, updating the branching policy $\pi(s_t)$ towards the optimal one basically means adjusting internal parameters (weights) of the function $Q(s_t, a_t)$ so that the magnitude of $\Delta Q(s_t, a_t)$ gets minimized.

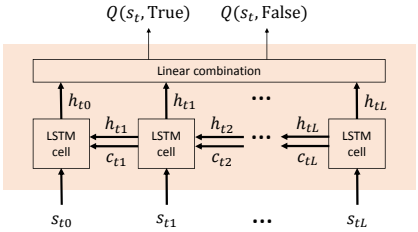


Fig. 5: Architecture of $Q(s_t, a_t)$ that is built using a special kind of ANNs, called a LSTM. Here, h_{tl} denotes a feature vector extracted by the LSTM cell, and c_{tl} is a cell state vector.

C. Q-network architecture

In practice, updating $Q(s_t, a_t)$ in (5) separately for each (s_t, a_t) is unattainable, since the state is a multi-dimensional integer vector and thus the number of possible states can be too large. Thus, a function approximator is commonly used to estimate the function $Q(s_t, a_t)$ with the limited number of observations for state-action pairs. Such a function approximator is typically a linear combination of representative features of (s_t, a_t) [18], [34]. However, recently an ANN-based non-linear function approximator is getting more attention in various applications, including Google's AlphaGo [35], [36].

PySE also represents $Q(s_t, a_t)$ by using an ANN-based function approximator, which we refer to as a **Q-network**. As shown in Figure 5, the Q-network of PySE is made using the long short term memory (LSTM) cell [37], [38], a special kind of recurrent neural networks (RNNs) that is capable of learning long-term dependencies between input values, and thus a good fit to handle the sequence structure of a state in PySE. The Q-network is created by applying the same LSTM cell recursively over elements of a state s_t in the order from s_{tL} to s_{t0}. The LSTM cell outputs a two-tuple, h_{tl} and c_{tl}, which are fed forward to next computations. Here, h_{tl} is a feature vector extracted from a sequence from s_{tl} to s_{tL}, thereby containing dependency information in there. The c_{tl} denotes a cell state vector, which is a kind of memory that selectively remembers computation results so far. Since the interior structure of the LSTM cell is not the focus of this work, interested readers may refer to [37], [38] for detail. The output of the Q-network is a linear combination of the feature vectors, written as the following matrix multiplication:

$$\begin{bmatrix} Q(s_t, \texttt{True}) \\ Q(s_t, \texttt{False}) \end{bmatrix} = W_q \begin{bmatrix} h_{t0} & h_{t1} & \cdots & h_{tL} \end{bmatrix}^T \quad (7)$$

where W_q is a weight matrix, and $\begin{bmatrix} h_{t0} & h_{t1} & \cdots & h_{tL} \end{bmatrix}$ denotes a concatenation of all feature vectors. Hereafter, we will use a **weight set** W to refer to all the weights within the Q-network collectively that include the weights within the LSTM cell in addition to W_q.

As mentioned earlier, updating the branching policy $\pi(s_t)$ means basically adjusting the weight set W to minimize the magnitude of $\Delta Q(s_t, a_t)$. Such minimization can be con-

Algorithm 1 Basic mode of PySE

```
1:  procedure SYMBOLIC EXECUTION
2:      for t from 1 to T do
3:          Choose a number u randomly over [0, 1].
4:          if u < ε then
5:              Choose a_t randomly.                    ▷ ε-greedy.
6:          else
7:              a_t = π(s_t).
8:          Execute a_t, and observe r_t and s_{t+1}.
9:          if the experience e_t = (s_t, a_t, r_t, s_{t+1}) is new then
10:             Add e_t in E.
11:     Delete old experiences in E to keep |E| ≤ N_e.
12: procedure POLICY UPDATE
13:     Sort experiences in E in a random order.
14:     for i from 1 to |E| do
15:         Read the i-th experience from E.
16:         Update weights in W according to (8).
```

ducted by various optimization techniques. For example, if we use the stochastic gradient descent, any weight $w \in W$ is updated until convergence as:

$$w \leftarrow w - \alpha \frac{\partial}{\partial w} \Delta Q(s_t, a_t). \tag{8}$$

D. Basic mode and its issue

Algorithm 1 illustrates two components of PySE that iterate in a basic mode. SYMBOLIC EXECUTION in line 1 corresponds to the step to run a program symbolically, and is mainly responsible for collecting a transition sample of (s_t, a_t, r_t, s_{t+1}), which we call an **experience**. At the t-th branch condition, PySE takes a branch determined by a_t, observes r_t and s_{t+1}, and stores the four-tuple experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in an **experience set** E. Here, we add new experiences only in the the experience set E (line 9). Thus, E contains unique experiences only. This prevents the Q-network from being trained with bias to particular experiences due to duplication, since all experiences can now be used the same number of times for weight updates. The symbolic execution lasts until a program terminates normally, or it encounters an infeasible path condition, *i.e.*, $r_t = P$ in (2), which is indicated by T in line 2. Note that by the ϵ-greedy strategy, action a_t is decided randomly with probability ϵ instead of $a_t = \pi(s_t)$ in all executions. This is how PySE explores a new execution path that is not specified by the branching policy $\pi(s_t)$. As in line 11, old experiences in E are deleted in an oldest-first manner when $|E|$, denoting the number of experiences in E, becomes larger than a threshold N_e. This prevents a particular experience from being used too many times for updating W over iterations, and the set E from being ever increasing.

POLICY UPDATE in line 12 is the step to update $\pi(s_t)$ using the experiences gathered in E by SYMBOLIC EXECUTION. For each experience, which has sufficient information to determine the value of $\Delta Q(s_t, a_t)$ in (6), the weight set W of the Q-network is updated according to (8). This update makes $\pi(s_t)$ converge to the optimal branching policy $\pi^*(s_t)$, and in a long-term trend, increases the expected length of a feasible execution path determined by $\pi(s_t)$, which is equivalent to $\mathbb{E}(\sum_{k=1}^{T} r_k)$. Note that by line 13, we are randomizing the order of experiences in E. This is to break strong correlation between timely-consecutive experience samples and thus to reduce the variance of the updates [36]. The number of experiences in E, *i.e.*, $|E|$, decides how many times we update W in a call to POLICY UPDATE. In lines 15 and 16, more than one experience can be used at a time to update W in what is called a mini-batch method, which may result in smoother convergence.

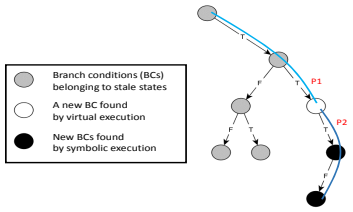


Fig. 6: Unique Path Finder that discovers a prefix (P1) of a brand-new execution path by virtual execution, which is a run over a computation tree built by observed experiences. Symbolic execution that follows is guided by the prefix P1 and finds out the remaining (P2) of the new execution path.

The basic mode of PySE may work well for programs whose optimal branching policy $\pi^*(s_t)$ is simple to describe, *e.g.*, the optimal action is always `True` for all t. However, it may take a painfully long time to learn $\pi^*(s_t)$ if there is no obvious pattern in the sequence of optimal actions over t. As shown in Figure 1c, a Dijkstra algorithm using a min-priority queue, for example, falls in this category. After a thorough investigation, we have found that this slowness in learning is mainly due to inefficiency of the ϵ-greedy strategy in exploring a new path. In detail, the problem is that since ϵ is small, the chance of finding a new execution path quickly decreases over iterations, resulting in many duplicated execution paths. When an execution path is what we have already observed, we cannot gather any new experience, thereby wasting time. Note that a large value of ϵ is not a fix to this issue, since then $\pi(s_t)$ is rather close to a random policy, *i.e.*, no use of intelligence built up in the Q-network. The ϵ-greedy with ϵ annealed down over iterations from a large value to a small one may alleviate the situation, but still it cannot prevent paths from being duplicated.

E. Unique Path Finder

Unique Path Finder (UPF) is a component of PySE that is intended to resolve the issue mentioned in Section IV-D. In other words, UPF attempts to help us gather at least one new experience in each symbolic execution step. Towards this end, what UPF does is **virtual execution**. Here, the virtual execution is defined as a sequence of state transitions using $\pi(s_t)$ with an ϵ-greedy strategy over an observed computation tree, which means a computation tree built up by all of observed experiences. Namely, the virtual execution is not an execution of a real program, but a simulation of state transitions among states that have been already observed,

Algorithm 2 Advanced mode of PySE (complete version)

```
1: Initialize an experience set E at start.
2: procedure SYMBOLIC EXECUTION
3:     for t from 1 to T do
4:         if ā_t exists then
5:             a_t = ā_t.                ▷ guided by the founding of UPF.
6:         else
7:             Choose a number u randomly over [0, 1].
8:             if u < ε then
9:                 Choose a_t randomly.
10:            else
11:                a_t = π(s_t).
12:        Execute a_t, and observe r_t and s_{t+1}.
13:        if the experience e_t = (s_t, a_t, r_t, s_{t+1}) is new then
14:            Add e_t in E.
15:    Delete old experiences in E to keep |E| ≤ N_e.
16: procedure POLICY UPDATE
17:    ▷ POLICY UPDATE is the same as in Algorithm 1.
18: procedure UNIQUE PATH FINDER (UPF)
19:    for i from 1 to N_search do
20:        Do virtual execution (VE) with ε-greedy strategy.
21:        if a new state found by VE then
22:            Report a prefix (ā_1, ā_2, ...).
23:            Break.
24:        if i = N_search/2 then
25:            Double the value of ε.
```

which we call **stale states**. Such a simulation takes negligible time to run.

As illustrated in Figure 6, a trial of virtual execution successfully finishes when it finds a new branch condition, equivalently a new state that has not yet been observed, which also means finding a new experience. When virtual execution ends up with an already-known path (*i.e.*, termination at a stale state), UPF can repeat virtual execution until successful, since the running cost of virtual execution is trivial.

The path from the starting branch condition to the new one found by virtual execution is what we call a **prefix** of a new execution path. Since virtual execution does not use a real program, UPF cannot describe the new branch condition itself. Instead, UPF represents the prefix by a sequence of actions in there. At the following iteration cycle, symbolic execution is first guided by this sequence of actions, and the new branch condition is then actualized. From that branch condition on, normal symbolic execution is performed, which completes the rest of the brand-new execution path.

F. A complete version of PySE

Algorithm 2 describes all components of PySE in its complete version, which we call an **advanced mode**, where each component iterates in the order shown in Figure 3. In line 4, $\bar{a}_t$ denotes action in the prefix of a new path found by UPF. When SYMBOLIC EXECUTION is guided by such actions at its beginning phase, the state s_t in the first execution of line 11 is guaranteed to be brand-new.

UNIQUE PATH FINDER (UPF) repeats virtual execution with a given value of ϵ, at most N_{search} times, until it can find a prefix of a new execution path, as in lines 21 to 23. The found prefix is used by SYMBOLIC EXECUTION in the next iteration cycle. If UPF cannot find a prefix of any new path within N_{search} times of virtual executions, SYMBOLIC EXECUTION that follows will start directly at line 7. In order to increase the chance of finding the prefix, the value of ϵ is doubled at $i = N_{search}/2$, as in line 25.

V. EXPERIMENTAL RESULTS

Our evaluation answers the following research questions:

RQ 1: How efficiently does PySE find the worst-case path, compared to exhaustive search, WISE, and SPF-WCA?

RQ 2: Is the advanced mode of PySE better than the basic mode?

RQ 3: Is pre-training the Q-network helpful in reducing the search time?

RQ 4: Does the history length L affect the search time?

Benchmark programs: We evaluate PySE by using it to find the worst-case at varying scales for two real-world applications and five micro-benchmark programs. We chose these programs as they are common benchmarks in stress testing literature [13], [14].

- Biopython `parewise2`: Biopython is a Python tool set for biological computation. In this case study, we applied PySE to its `pairwise2.align.localxx` module, which determines similar regions between two strings of nucleic acid sequences or protein sequences, using Smith-Waterman algorithm [39].
- GNU `grep`: This is a tool for a key string search over a given text using Boyer-Moore algorithm [41]. We used a Python implementation in [42] with a fixed key string of length 3 and the text string made of N symbolic variables. Boyer-Moore algorithm has one branch point, and in this setup, the sequence of the worst-case branching decisions is (`True`, `True`, `False`) for $N = 3, 4, 5$, (`True`, `True`, `False`,`True`, `True`, `False`) for $N = 6, 7, 8$, and so on (refer to Figure 1b). Namely, it is the i times repetition of (`True`, `True`, `False`) for $N = 3i, 3i + 1, 3i + 2$. This kind of discontinuity over scales occurs due to the so-called good suffix rule of Boyer-Moore algorithm [41].
- Micro-benchmarks: We consider insertion sort (the most common benchmarks for stress testing), priority queue (enqueueing), binary search tree (building a tree), traveling salesman, and Dijkstra (using a min-priority queue).

All experiments are conducted on a Dell PowerEdge R320 server of 12 Intel Xeon processors at 2.4GHz, and 16GB memory. Unless otherwise stated, we set $N_{search} = 200$, $N_e = 5000$, and $L = 2$. In the Q-network, the lengths of h_{tl} and c_{tl} are both set to 2. Minor variation of ϵ does not give an observable change in the result, and the trend by the major variation is well known in the reinforcement learning literature, which we also briefly mentioned in Section IV-D. Thus, in our experiments, we fix $\epsilon = 0.1$.

Branching pattern classes: Depending on the worst-case behavior, we categorize the benchmark programs into two classes. Class 1 programs (listed in Table I) are the ones where

TABLE I: Results for Class 1 programs. Here, N is the number of symbolic variables as inputs, 'Paths' means the number of paths to search including infeasible ones. It reports the total number of paths explored until finding the worst case. 'Time' is the corresponding elapsed time in a 'min:sec' format. The symbol '-' denotes that a corresponding experiment could not finish in 1,000 minutes.

	(N, longest path length)		(3,9)	(4,12)	(5,15)	(10,30)	(20,60)	(30,90)	(100,300)
Benchmark 1: **Biopython `parewise2`**: Smith-Waterman [39]	Exhaustive search	Paths	127	511	2047	-	-	-	-
		Time	0:04	0:18	1:14	-	-	-	-
	WISE	Paths	1	1	1	1	1	1	1
		Time	0:00	0:00	0:00	0:00	0:00	0:00	0:01
	PySE	Paths	1	1	1	1	1	1	2
		Time	0:02	0:02	0:02	0:02	0:02	0:02	0:13
	(N, longest path length)		(3,3)	(4,6)	(5,10)	(10,45)	(20,190)	(30,435)	(100,3875)
Benchmark 2: **Insertion sort**	Exhaustive search	Paths	6	24	120	-	-	-	-
		Time	0:00	0:00	0:01	-	-	-	-
	WISE	Paths	1	1	1	1	1	1	1
		Time	0:00	0:00	0:00	0:00	0:00	0:00	0:09
	PySE	Paths	2	1	4	4	2	5	2
		Time	0:11	0:02	0:30	0:30	0:30	3:19	23:04
	(N, longest path length)		(3,2)	(4,4)	(5,6)	(10,19)	(20,54)	(30,94)	(100,480)
Benchmark 3: **Priority queue** (enqueue) [40]	Exhaustive search	Paths	4	12	36	20736	-	-	-
		Time	0:00	0:00	0:01	21:02	-	-	-
	WISE	Paths	1	1	1	1	1	1	1
		Time	0:00	0:00	0:00	0:00	0:00	0:00	0:00
	PySE	Paths	1	1	1	2	2	25	47
		Time	0:02	0:02	0:02	0:10	0:12	3:50	11:19
	(N, longest path length)		(3,3)	(4,6)	(5,10)	(10,45)	(20,190)	(30,435)	(100,4950)
Benchmark 4: **Binary search tree** (build)	Exhaustive search	Paths	6	24	120	-	-	-	-
		Time	0:00	0:00	0:01	-	-	-	-
	WISE	Paths	1	1	1	1	1	1	1
		Time	0:00	0:00	0:00	0:00	0:00	0:00	12:40
	PySE	Paths	1	1	2	2	2	2	2
		Time	0:02	0:02	0:11	0:15	0:21	1:08	21:02

regardless of scales, all the branch points have a fixed decision (`True` or `False`) in the worst case, as in Figure 1a. These are the programs where WISE is effective, and SPF-WCA works exactly the same as WISE. Class 2 programs (listed in Table II) are the one where some or all of branch points have a non-deterministic decision in the worst case, as in Figures 1b and 1c. In Class 2, the worst-case-leading decision at a branch point can change depending on the scale (N), or the time (t) that the branch point is visited. WISE cannot handle Class 2 programs efficiently. However, SPF-WCA can be effective for some of them, *i.e.*, when there is an obvious pattern in alternation between `True` and `False` as in Figures 1b.

Comparison in Class 1 programs (RQ1): Table I shows comparison results in Class 1 programs. Here, SPF-WCA is functionally the same as WISE [14] so we omit the result for SPF-WCA. For each program, WISE learned the worst-case branching policy using the smallest and the second smallest scales shown on the table. The learning time of WISE is not included in the table. PySE's result is the median among five sets of Monte Carlo experiments, for which the weight set W of the Q-network is randomly initialized.

We can first see that exhaustive search faces exponentially growing path diversity and search time so it becomes quickly infeasible to finish its search within a limit (as which we set 1,000 minutes). In contrast, WISE predicts the worst-case path using its branching policy and thus searches one single path only, achieving the search time that is negligible. In the same situation, PySE can also learn the worst-case path within a few trials, since the branching pattern is simple. However, after each trial, PySE needs time to update the branching policy $\pi(s_t)$ (*i.e.*, training the Q-network). For this reason, the search time of PySE is longer than that of WISE, but we can say that it is still trivial compared to exhaustive search.

Comparison in Class 2 programs (RQ1): The same kind of comparison experiments are done in Class 2 programs and Table II shows the results. For this table, WISE and SPF-WCA learned the worst-case branching policy using exhaustive search on the smallest and the second smallest scales shown on the table unless otherwise stated, and the learning time of WISE and SPF-WCA is not included in the table, as before. As a default value, SPF-WCA used the history of length 2, like $L = 2$ in PySE, meaning that SPF-WCA makes a branching decision considering the two previous branching decisions and conditions. In PySE, the weight set W of the Q-network is initialized with the one that was pre-trained at scale $N = 5$ for benchmark 5, and $N = 6$ for benchmarks 6 and 7, unless otherwise stated. This is intended to speed-up the learning if possible, and the effect of loading the pre-trained weight set at the beginning can be seen in more detail in Figure 7b.

In the benchmark 5, GNU `grep`, since the worst-case branching decision is not deterministic at a branch point, WISE cannot predict the worst-case path, and has to search all the possible cases, like exhaustive search. In the meantime, SPF-WCA may handle this situation effectively. Since SPF-WCA considers history, it can learn what decision should be taken to lead to the worst case when the previous decision sequences are (`True`, `True`), (`True`, `False`), or (`False`,`True`). However, this is only possible when SPF-WCA learns the worst-case branching policy on scales over 6 (*i.e.*, $N \geq 6$), because up to $N = 5$, the sequence of the worst-case branching decisions is always (`True`, `True`, `False`) so that it cannot learn what will follow after (`True`, `False`) on a larger scale, for example. The table shows that for this reason, SPF-WCA has to do exhaustive search through some part of the execution tree, and the number of paths to search increases exponentially over scales. Note that as mentioned, SPF-WCA can predict the worst-case path at large scales with a single trial, if trained with $N \geq 6$. However, such critical choices of training scales requires deep understanding

TABLE II: Results for Class 2 programs. The symbol 'x' denotes the prediction of the worst-case path is wrong. The symbol '?' on the longest path length means that the number is not double-checked, since other solutions could not find the worst-case path within the limit.

	(N, longest path length)		(3,3)	(4,3)	(5,3)	(10,9)	(20,18)	(30,30)	(100,99)
Benchmark 5: **GNU `grep`**: Boyer-Moore [41]	Exhaustive search	Paths	4	4	4	40	1093	88573	-
		Time	0:00	0:00	0:00	00:01	00:31	43:39	-
	WISE	Paths	4	4	4	40	1093	88573	-
		Time	0:00	0:00	0:00	00:01	00:32	44:24	-
	SPF-WCA trained at $N = 3, 4$	Paths	1	1	1	9	243	19683	-
		Time	0:00	0:00	0:00	00:00	00:07	10:20	-
	SPF-WCA trained at $N = 6, 7$	Paths	1	1	1	1	1	1	1
		Time	0:00	0:00	0:00	00:00	00:00	00:00	00:00
	PySE pre-trained at $N = 5$	Paths	2	2	2	2	2	3	276
		Time	0:11	0:11	0:11	0:11	0:12	0:20	48:21
	PySE pre-trained at $N = 10$	Paths	2	2	2	1	2	3	82
		Time	0:11	0:11	0:11	0:02	0:12	0:20	13:03

	(N, longest path length)		(3,4)	(6,12)	(10,28?)	(15,55?)
Benchmark 6: **Traveling salesman** [43]	Exhaustive search	Paths	8	728	-	-
		Time	0:00	0:22	-	-
	WISE	Paths	8	728	-	-
		Time	0:00	0:23	-	-
	SPF-WCA	Paths	1	1	x	-
		Time	0:00	0:00	2:18	-
	PySE	Paths	2	7	102	1501
		Time	0:11	0:56	41:10	204:16

	(N, longest path length)		(6,5)	(12,29)	(20,72)	(30,126?)
Benchmark 7: **Dijkstra** (with a min-priority queue) [15]	Exhaustive search	Paths	9	837	-	-
		Time	0:00	0:10	-	-
	WISE	Paths	7	518	91275	-
		Time	0:00	0:22	114:13	-
	SPF-WCA	Paths	3	269	x	-
		Time	0:00	0:10	144:05	-
	PySE	Paths	4	114	1020	5534
		Time	0:29	16:46	172:42	819:27

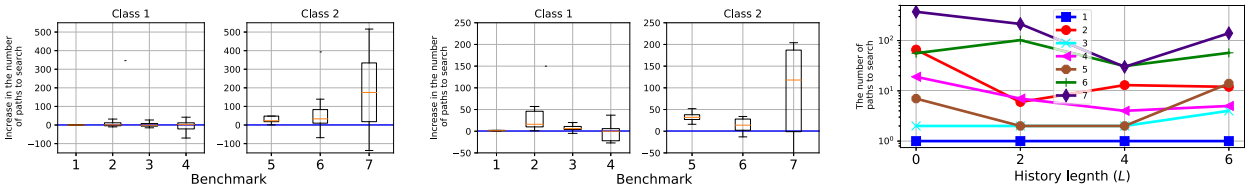
of a target program, and thus, it can be said a drawback of SPF-WCA as a black-box testing tool. In contrast, PySE is flexible in responding to such discontinuity over scales, since it learns the worst-case path directly at the target scale. Looking at the difference caused by the scale that pre-training is done at, PySE can also get a little more benefit when pre-trained with more patterns of the branching decision sequence. However, even if PySE encounters a pattern that has not seen in the pre-training, it can update the branching policy from the experiences collected at the target scale, and achieve faster search time than WISE and SPF-WCA at large scales.

For benchmark 6, traveling salesman algorithm [43], there exists one branch point, but it shows many different worst-case paths (*e.g.*, there are 96 worst-case paths of the same length at $N = 6$), where the sequences of the worst-case branching decisions are all irregular alternation between `True` and `False`. Thus, WISE cannot learns a fixed branching policy and acts like exhaustive search. On the other hand, SPF-WCA collects 7 kinds of histories of length 2 and fixes the branching decisions at 6 of them, resulting in the search time that is within the limit at $N = 10$. However, this causes a wrong prediction to the worst-case (we could check it since PySE found paths that are at least longer than the prediction of SPF-WCA). This might be because the training scales are too small, but we could not add the next possible scale in the training, since exhaustive search at scale $N = 10$ could not finish within the time limit. At $N = 10$ and $N = 15$, we can see that only PySE finished the search within the time limit. PySE found a longer path than SPF-WCA (the path length is reported in the table), but we could not check if the found path is the true worst case, since no other solutions could confirm it within the time limit.

Benchmark 7, Dijkstra using a min-priority queue [15] has three branch points, and WISE can fix the worst-case branching decision at only one of them (corresponding to the marker ■ in Figure 1c). Thus, it could reduce the number of paths to search a little, compared to exhaustive search, but it was not effective enough to handle large scales. SPF-WCA collects 17 kinds of histories of length 2 and fixes the branching decisions at three of them. However, we can still see that search time increases exponentially over scales. In addition, SPF-WCA predicts a wrong worst-case path in this case as well at $N = 20$ (we could check it because SPF-WCA predicted a shorter path than WISE). In sum, WISE and SPF-WCA reduce the search scope partially, but search time still increases exponentially over scales, leaving $N = 30$ uncovered. In contrast, we can see that PySE can still handle $N = 30$, although it takes a long time to finish its search, due to complex branching patterns.

Basic mode vs. advanced mode (RQ2): Figure 7a shows how many more paths the basic mode of PySE has to search until finding the worst-case path, compared to the advanced mode. We can first notice that in Class 1 programs, the basic mode of PySE works almost the same as the advanced mode. This means that when the branching policy has a simple pattern, the basic mode is as good as the advanced mode. However, in Class 2 programs, the speed-up benefit of the advanced mode becomes significant. To learn the complex branching patterns of Class 2 programs, PySE needs to collect various unique experiences. In such a situation, controlled exploration to new paths by UPF indeed saves time as intended.

Effect of pre-training (RQ3): Since PySE's state representation is scale-independent, PySE can benefit at some cases from reusing the pre-trained Q-network at different scales. Figure 7b shows an example of how many trials we can save in search by loading the pre-trained Q-network, instead of random initialization. By looking at the median value, we can see that Class 1 programs where the branching policy has a simple pattern do not benefit from using loading pre-trained Q-network. However, in Class 2 programs, we can



(a) Effect of advanced mode. (b) Effect of pre-training. (c) Effect of history length.

Fig. 7: Effects of internal parameter choices in PySE. Here, $N = 10$ for all benchmarks except that $N = 12$ for benchmarks 7 and 8: (a) Increase in the number of paths to search until finding the worst-case path when we switch from the advanced mode of PySE to its basic mode. (b) Increase in the number of paths to search until finding the worst-case path when switching from the case we use the pre-trained Q-network to the case we initialize Q-network with random values. We used the next smallest scale in Tables I and II for pre-training. (c) The number of paths to search for each benchmark, depending on the history length. Here, the number of paths to search is the median of five sets of experiments.

notice the difference, especially at benchmark 7. This result implies that when the target program has complex branching patterns, PySE can be faster with pre-training at a lower scale. **Effect of history legnth L (RQ4)**: Figure 7c shows the effect of the history length L in the search time for each benchmark. We can first notice that there is no common trend: every benchmark has different optimal history length. Since $L = 2$ is not the best for some benchmarks, we can know that the results of PySE in Tables I and II can be improved with other choice of L. Some benchmarks (*e.g.*, benchmark 4) reduce search time with non-zero L. Some benchmarks (*e.g.*, benchmark 5) show that too long history may rather increase the search time. However, note that PySE can still find the worst-case with any value of L, which is different from SPF-WCA, where the exact number of the history length affects the correctness of worst-case path prediction.

VI. Discussion

We can see from Tables I and II that the search time of PySE increases according to input scales even for Class 1, albeit slowly. This is mainly due to fact that computation time of a constraint solver increase with a scale. Thus, the scale that PySE can handle is somehow bounded in practice, although it can be much larger than $N = 100$ for Class 1. Our future work will be overcoming this limitation by devising a way to reduce the number of times that PySE invokes the constraint solver.

Since PySE approximates $Q(s_t, a_t)$ by the Q-network, the convergence to the optimal branching policy $\pi^*(s_t)$, *i.e.*, finding the path of the worst-complexity is not always guaranteed, although most of our experiment results are confirmed to be the true worst case. However, as can be seen in Table II, PySE can still be used as a black-box tool to find out a long enough path within a reasonable time bound in the cases where other solutions cannot.

PySE holds any inherent limitations of symbolic execution (such as the difficulty in handling environment interactions), but common remedies to them (such as redirecting access to a pre-defined environment model) are also applicable to PySE.

In this work, we define the complexity of a program as the length of an execution path. Different metrics such as execution time or memory usage can be considered in PySE easily, by changing definition of the reward function r_t in (2) accordingly. For example, when the worst case of memory usage is the matter, r_t can be defined as the amount of memory increase caused by action a_t.

VII. Conclusion

In this paper, we proposed PySE that can generate inputs that cause a program to execute in the worst case mode. This tool can thus be used for stress testing. PySE uses symbolic execution to run a program and collects behavioral information. PySE then updates a branching policy using the collected behaviors based on a reinforcement learning framework. By iterating the symbolic execution and policy update, PySE gradually increases the length of an execution path towards a path of the worst-case complexity. PySE limits the search scope to variations of the path determined by the branching policy that is updated at each iteration. By this, PySE can prune out most of unnecessary paths and handle a large scale program, where exhaustive search is infeasible due to the path explosion problem. PySE's learning procedure is fully automatic and it can learn a complex or irregular pattern of branching decisions, which WISE-like algorithms cannot handle. We demonstrated that in various Python programs and scales, PySE can effectively find a path of worst-case complexity and has benefits against exhaustive search and WISE-like algorithms.

Acknowledgment

The authors would like to thank anonymous reviewers for their helpful comments and feedback. This work was partly supported by DOE Early Career Award DE-SC0010295, NSF awards CNS-1527262, CNS-1513197, CCF-115013 (CAREER) and CCF-1439126, and an unrestricted gift from Adobe Research. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] Z. M. Jiang and A. E. Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, Nov 2015.

[2] Wikipedia. Load testing. https://en.wikipedia.org/wiki/Load_testing, 2018. Accessed : 5-Oct-2018.

[3] Tse-Hsun Chen, Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP, pages 243–252, 2017.

[4] R. Gao, Z. M. Jiang, C. Barna, and M. Litoiu. A framework to evaluate the effectiveness of different load testing analysis techniques. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 22–32, April 2016.

[5] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[6] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, pages 150–153, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS/ETAPS, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS, 2008.

[10] Manuel Costa. Bouncer: Securing software by blocking bad input. In *Proceedings of the 2nd Workshop on Recent Advances on Intrusiton-tolerant Systems*, WRAITS, New York, NY, USA, 2008. ACM.

[11] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[13] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE, pages 463–473, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Kasper Luckow, Rody Kersten, and Corina Pasareanu. Symbolic complexity analysis using context-preserving histories. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 58–68, March 2017.

[15] Wyatt Lee Baldwin. Dijkstar 2.2 — PyPI - the Python Package Index. https://pypi.python.org/pypi/Dijkstar/2.2, 2009. Accessed : 5-Oct-2018.

[16] Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. Vrisha: Using scaling properties of parallel programs for bug detection and localization. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC, pages 85–96, New York, NY, USA, 2011. ACM.

[17] Bowen Zhou, Jonathan Too, Milind Kulkarni, and Saurabh Bagchi. Wukong: Automatically detecting and localizing bugs that manifest at large system scales. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC, pages 131–142, New York, NY, USA, 2013. ACM.

[18] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[19] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[20] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE, pages 475–485, 2018.

[21] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 2155–2168, New York, NY, USA, 2017. ACM.

[22] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, pages 254–265, New York, NY, USA, 2018. ACM.

[23] Wikipedia. Genetic algorithm. https://en.wikipedia.org/wiki/Genetic_algorithm, 2018. Accessed : 5-Oct-2018.

[24] Miguel Rocha and José Neves. Preventing premature convergence to local optima in genetic algorithms via random offspring generation. In Ibrahim Imam, Yves Kodratoff, Ayman El-Dessouki, and Moonis Ali, editors, *Multiple Approaches to Intelligent Systems*, pages 127–136. Springer Berlin Heidelberg, 1999.

[25] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA, pages 225–236, New York, NY, USA, 2009. ACM.

[26] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. S-looper: Automatic summarization for multipath string loops. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 188–198, New York, NY, USA, 2015. ACM.

[27] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 61–72, New York, NY, USA, 2016. ACM.

[28] H. J. Bang, T. H. Kim, and S. D. Cha. An iterative refinement framework for tighter worst-case execution time calculation. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC, pages 365–372, May 2007.

[29] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for wcet analysis. In *Perspectives of Systems Informatics*, pages 227–242, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[30] Kanak Mahadik, Christopher Wright, Jinyi Zhang, Milind Kulkarni, Saurabh Bagchi, and Somali Chaterji. Sarvavid: A domain specific language for developing scalable computational genomics applications. In *International Conference on Supercomputing*, ICS, pages 34:1–34:12, 2016.

[31] M. de Jonge and E. Visser. Automated evaluation of syntax error recovery. In *The 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 322–325, Sep. 2012.

[32] Charitha Saumya, Jinkyu Koo, Milind Kulkarni, and Saurabh Bagchi. XSTRESSOR: Automatic Generation of Large-Scale Test Inputs by Inferring Path Conditions. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2019.

[33] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[34] Jinkyu Koo, Xiaojun Lin, and Saurabh Bagchi. Rl-blh: Learning-based battery control for cost savings and privacy preservation for smart meters. In *The 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017.

[35] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602, 2013.

[37] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.
[38] Christopher Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. Accessed : 5-Oct-2018.
[39] Wikipedia. Smith-Waterman algorithm. https://en.wikipedia.org/wiki/Smith-Waterman_algorithm, 2018. Accessed : 5-Oct-2018.
[40] Queue A synchronized queue class. The Python Standard Library. https://docs.python.org/2/library/queue.html, 2017. Accessed : 5-Oct-2018.
[41] Wikipedia. Boyer-Moore string search algorithm. https://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm, 2018. Accessed : 5-Oct-2018.
[42] Huyen Tran. python-grep. https://github.com/heyhuyen/python-grep, 2018. Accessed : 5-Oct-2018.
[43] Dmitry. Suboptimal Travelling Salesman Problem (TSP) solver. https://github.com/dmishin/tsp-solver, 2017. Accessed : 5-Oct-2018.