PYSE: Automatic Worst-Case Test Generation by Reinforcement Learning

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 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 improvement, 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 1c), or “alternate between True and False decisions at a branch point in fixed patterns” (see Figure 1c). 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 \((N > N_0)\), where \(N\) is an input size variable and \(N_0\) 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 \[19\]. 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. No special consideration is given there to find out a path of the worst-case complexity.

WISE \[13\] came up with an efficient way to predict a path of the worst-case complexity at a large scale. It performs exhaustive search using symbolic execution on small scales, and learn 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 \[15\], 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.

---

\(^1\)The term PySE derives from Python and Symbolic Execution, and is pronounced like the end of “spice”.

\(^2\)Source code of PySE is available at http://x.y.z (anonymized).
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 \( a_t = \pi(s_t) \in \{\text{True}, \text{False}\} \), 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 \( B_1 \) in Figure 2 that defines the state at the branch condition \( C_1 \) 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:

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.

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.

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

\[
\pi(s_t) = \arg \max_{a_t} Q(s_t, a_t),
\]

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, \( Q(s_t, \text{True}) \) and \( Q(s_t, \text{False}) \). That is, \( \pi(s_t) = \text{True} \) if \( Q(s_t, \text{True}) > Q(s_t, \text{False}) \) and \( \pi(s_t) = \text{False} \) 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 \( Q(s, \text{True}) \) and \( Q(s, \text{False}) \) 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.

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.
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 $\epsilon$ (typically $\leq 0.1$), 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 $\epsilon$. Such exploration by random action with probability $\epsilon$ is called the $\epsilon$-greedy strategy in reinforcement learning literature [18].

To get a better understanding of the $\epsilon$-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 $\epsilon$ 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 $\epsilon$-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.

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:

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

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
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, $Q^*(s_t, a_t) = \max_a \mathbb{E} \left( \sum_{t=1}^{T} r_k | s_t, a_t \right)$. 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:

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

where $\max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) = 0$, by definition, when $s_{t+1}$ is the state of program termination. In (3), we do not know the value of $r_t + \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$ 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 $\alpha$ 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 $\alpha$. 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 $\alpha$ [18], [29]. 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:

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

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.

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], [29]. However, recently an ANN-based non-linear function approximator is getting more attention in various applications, including Google’s AlphaGo [30], [31].

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 [32], [33], 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 $s_{L}$ to $s_{0}$. 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 $s_{L}$ to $s_{0}$, 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 [32], [34] for detail. The output of the Q-network is a linear combination of the feature vectors, written as the following matrix multiplication:

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

![Figure 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.](image)
where $W_q$ is a weight matrix, and $[h_{t0} \ h_{t1} \ \cdots \ h_{tL}]$ 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 conducted by various optimization techniques. For example, if we use the stochastic gradient descent, any weight $w \in W$ is updated until convergence as:

$$w \leftarrow w - \alpha \frac{\partial}{\partial w} \Delta Q(s_t, a_t).$$  \hspace{1cm} (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 $\epsilon$-greedy strategy, action $a_t$ is decided randomly with probability $\epsilon$ 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.

**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 $u < \epsilon$ then
5:    Choose $a_t$ randomly. \hspace{1cm} \triangleright \ \epsilon$-greedy.
6:  else
7:    $a_t = \pi(s_t)$.
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 $|E| \leq N_e$.
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).

**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 $E[\sum_{k=1}^T r_k]$. 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 (8). 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.

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 6, 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 $\epsilon$-greedy strategy in exploring a new path. In detail, the problem is that since $\epsilon$ 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 $\epsilon$ 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 $\epsilon$-greedy with $\epsilon$ 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 $\epsilon$-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, 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 $a_t$ 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 $\epsilon$, 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 2. In order to increase the chance of finding the prefix, the value of $\epsilon$ 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?

Algorithm 2 Advanced mode of PySE (complete version)

1: Initialize an experience set $E$ at start.
2: procedure SYMBOIC EXECUTION
3: for $t$ from 1 to $T$ do
4: if $a_t$ exists then
5: $a_t = \bar{a}_t$. \quad \triangleright guided by the founding of UPF.
6: else
7: Choose a number $n$ randomly over $[0, 1]$.
8: if $n < \epsilon$ then
9: Choose $a_t$ randomly.
10: else
11: $a_t = \pi(s_t)$.
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 $E$ if keep $|E| \leq N_e$.
16: procedure POLICY UPDATE
17: \triangleright 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 $\epsilon$-greedy strategy.
21: if a new state found by VE then
22: Report a prefix ($\bar{a}_1, \bar{a}_2, \ldots$).
23: Break.
24: if $i = N_{search}/2$ then
25: Double the value of $\epsilon$.

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.

- Biopython
- Biopython (pairwise2.align.localxx module): This 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.

- GNU grep: This is a tool for a key string search over a given text using Boyer-Moore algorithm. We used a Python implementation in [37] 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 15). 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.

- Micro-benchmarks: We consider insertion sort (the most common benchmarks for stress testing), priority queue (enqueuing), 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.
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.

<table>
<thead>
<tr>
<th>Benchmark 1: Biopython pairwise Smith-Waterman</th>
<th>( (N, \text{longest path length}) )</th>
<th>Exhaustive search</th>
<th>WISE ( (\text{SE}) )</th>
<th>PySE ( (\text{SE}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paths ( a )</td>
<td>Paths ( a )</td>
<td>Paths ( a )</td>
<td>Paths ( a )</td>
<td>Paths ( a )</td>
</tr>
<tr>
<td>Time ( t )</td>
<td>Time ( t )</td>
<td>Time ( t )</td>
<td>Time ( t )</td>
<td>Time ( t )</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(1,1)</td>
<td>(1,1)</td>
<td>(1,1)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(1,2)</td>
<td>(1,2)</td>
<td>(1,2)</td>
<td>(1,2)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

memory. Unless otherwise stated, we set \( N_{\text{search}} = 200 \), \( N_e = 5000 \), and \( L = 2 \). In the Q-network, the lengths of \( h_{tl} \) and \( c_{tl} \) are both set to 2. Minor variation of \( \epsilon \) 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 and II) are the ones where 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, 7b, and 11c. 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, 7b, and 11c.

Comparison in Class 1 programs (RQ1): Table I shows comparison results in Class 1 programs. Here, WISE 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 scale \( 2 \), 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.
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.

<table>
<thead>
<tr>
<th>Benchmark: Gnu grep, Boyer-Moore</th>
<th>SPF-WCA trained at N = 3, 4</th>
<th>SPF-WCA pretrained at N = 6, 7</th>
<th>SE</th>
<th>WISE</th>
<th>PySE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N, longest path length)</td>
<td>Exhustive search</td>
<td>Paths</td>
<td>Time</td>
<td>Paths</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>Paths</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For benchmark 6, traveling salesman algorithm [38], 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 learn 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 1G). 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 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 length 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 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.