Learning to Explore Paths for Symbolic Execution

Jingxuan He, ETH Zurich, Switzerland
jingxuan.he@inf.ethz.ch

Gishor Sivanrupan, ETH Zurich, Switzerland
gishors@student.ethz.ch

Petar Tsankov, ETH Zurich, Switzerland
petar.tsankov@inf.ethz.ch

Martin Vechev, ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

ABSTRACT
Symbolic execution is a powerful technique that can generate tests steering program execution into desired paths. However, the scalability of symbolic execution is often limited by path explosion, i.e., the number of symbolic states representing the paths under exploration quickly explodes as execution goes on. Therefore, the effectiveness of symbolic execution engines hinges on the ability to select and explore the right symbolic states.

In this work, we propose a novel learning-based strategy, called LEARCH, able to effectively select promising states for symbolic execution to tackle the path explosion problem. LEARCH directly estimates the contribution of each state towards the goal of maximizing coverage within a time budget, as opposed to relying on manually crafted heuristics based on simple statistics as a crude proxy for the objective. Moreover, LEARCH leverages existing heuristics in training data generation and feature extraction, and can thus benefit from any new expert-designed heuristics.

We instantiated LEARCH in KLEE, a widely adopted symbolic execution engine. We evaluated LEARCH on a diverse set of programs, showing that LEARCH is practically effective: it covers more code and detects more security violations than existing manual heuristics, as well as combinations of those heuristics. We also show that using tests generated by LEARCH as initial fuzzing seeds enables the popular fuzzer AFL to find more paths and security violations.

CCS CONCEPTS
- Software and its engineering → Software testing and debugging
- Security and privacy → Software security engineering

KEYWORDS
Symbolic execution; Fuzzing; Program testing; Machine learning

1 INTRODUCTION
Symbolic execution [18, 44] is a promising program analysis technique widely used in many security-related tasks, such as analyzing protocol implementations [23, 24, 53], validating hardware design [76], securing smart contracts [49, 55], and detecting cache timing leaks [36]. Most prominently, symbolic execution has been extensively used for automatic test generation to exercise program paths and identify security violations [8, 16, 17, 22, 68], and is by now an established industrial practice for software testing at Microsoft [34], IBM [6], NASA [52], and other organizations.

At a high level, symbolic execution works by representing program inputs as symbolic variables, exploring program paths symbolically, and collecting path constraints that capture conditions over the input variables that must hold to steer the program along a given path. These constraints can be fed into an external constraint solver to construct a concrete test case. The common goal for symbolic execution tools is to generate a test suite that achieves high code coverage over the program’s statements within an allocated time budget [16, 52, 74].

Key challenge: path explosion. While powerful, symbolic execution is expensive and difficult to scale to large, real-world programs due to the so-called path explosion problem [18]. That is, at each program branch, the (symbolic) state for a given path is forked into two separate states. As a consequence, the number of states is exponential in the number of branches and quickly explodes as the execution reaches deep branches. To cope with this challenge, symbolic execution tools need an effective mechanism to select and execute promising states that achieve the highest coverage and cost the least execution time, while avoiding expensive states that do not improve coverage.

Unfortunately, constructing an ideal state selection is not possible because at the time of picking a state to explore we do not know whether it will indeed improve coverage at a reasonable cost. Ultimately, this decision depends on how the selected state would...
Unfold as we continue executing it as well as on state selection decisions we make in the future. Due to this fundamental limitation, symbolic execution tools rely on manually designed heuristics, used as a proxy for the ideal selection, that select states based on properties such as instruction count [16], subpath count [48], constraint search depth [52], and hand-crafted fitness [74]. As a result, even though designed by experts, those heuristics can easily get stuck in program parts favoring the measured property and fail to reach other relevant parts that, if covered, would improve code coverage and may identify critical security violations.

**LEARCH: a learning-based state selection strategy.** In this work, we propose a new *data-driven* approach, called **LEARCH** (short for Learning-based **S**earch), for learning a state selection strategy that enables symbolic execution tools to efficiently explore the input program. The key idea is to leverage a machine learning regression model that, for each state, estimates a *reward* that directly captures the core objective of the tool — improving more coverage while spending less time producing concrete tests. Based on this model, **LEARCH** selects the state with the highest estimated reward, as opposed to relying on manual heuristics used as a proxy to maximize the tool’s objective. Importantly, the construction of **LEARCH** utilizes the knowledge of existing heuristics and can benefit from any advances in the invention of new heuristics.

**LEARCH** is constructed using an *iterative learning procedure*, as illustrated at the top of Figure 1. At each iteration, we first run symbolic execution on a set of training programs. Notably, instead of exploring states uniformly at random, we leverage different state selection strategies (e.g., manual heuristics at iteration 1) to generate a diverse set of tests. Then, for each explored state in the generated tests, we extract a set of high-level features (including the properties used by the heuristics) and calculate a reward based on the overall coverage improvement and time spent exploring the state. This results in a supervised dataset that captures the behaviors of the strategies used in the previous step. Finally, we construct a learned strategy by training a regression model to achieve a small loss on the supervised dataset so that the model can make accurate estimations for the reward. The strategy learned at the current iteration is used to add new supervised data in the next iterations to create additional learned strategies. At inference time (bottom of Figure 1), given an unseen program, we run multiple symbolic execution instances with the learned strategies to generate effective tests used to exercise the program and report security violations.

**Instantiation and evaluation.** We instantiated **LEARCH** on the most popular symbolic execution engine KLEE [16]. We evaluated **LEARCH** on a diverse set of programs, including 52 *coreutils* programs and 10 real-world programs. Our results demonstrate that **LEARCH** is practically effective: it consistently produced more code coverage (e.g., >20%) and detected more security violations (e.g., >10%) than existing manual heuristics [16, 48], as well as the combinations of individual heuristics. Moreover, we used tests generated by KLEE as initial seeds to run a popular fuzzer, AFL [1, 29]. The initial seeds from **LEARCH** helped AFL to trigger more paths and security violations than the manual heuristics.

**Main contributions.** Our main contributions are:
- **LEARCH**, a new learning-based state selection strategy for symbolic execution. (Section 3)
- A novel learning framework for constructing multiple learned strategies. (Section 4)
- A complete instantiation of **LEARCH** on the popular symbolic test generator KLEE. (Section 5)
- An extensive evaluation on a diverse set of programs and various tasks, demonstrating that **LEARCH** is practically effective and outperforms existing manually designed heuristics. (Section 6)

2 MOTIVATION FOR LEARNING

In this section, we motivate the use of learning for selecting symbolic states by analyzing the results of running KLEE [16] on the 52 *coreutils* programs used as one of the test sets in our evaluation. The time limit of KLEE was 1h, and the memory budget was 4GB.

**Large number of candidate states.** When symbolically executing an input program, KLEE usually forks a large number of states due to branching behaviors such as if-else statements, resolving function pointers, and resolving memory allocation sizes. To measure the number of states produced by KLEE, we ran the tool using the random path search heuristic (RPS) on the 52 *coreutils* programs and calculated the average number of candidate states at the selection steps, and show the results in Figure 2. For most programs, the number of states produced by KLEE is huge: 28 programs have on average 10k–100k candidate states for selection. For programs larger than *coreutils*, the number of candidate states could be even larger. The enormous search space motivates the need for constructing an effective, fine-grained strategy able to pick promising states instead of relying on simple and crude heuristics.

**Limitations of existing manually designed heuristics.** Existent heuristics are random or manually designed by experts and typically depend on certain property of the states [16, 48]. They often get stuck in program parts favoring the property but fail to explore other parts. We ran KLEE with a set of existing heuristics and present the average line coverage of the top three heuristics (RPS, nurs::depth, and sgs) in the Venn diagram in Figure 3(a). All three heuristics achieved ∼540 line coverage. However, no single heuristic significantly outperformed the others. Importantly, the heuristics have non-comparable performance and covered different parts of the program: 499 lines were covered by all heuristics, but the rest 86 lines were covered by different heuristics. Similarly, the capability of detecting security violations (UBSan violations [4] in our work) differs across the heuristics, as shown in Figure 3(b).

---

*LEARCH* is publicly available at https://github.com/eth-sri/learch.
3 SYMBOLIC EXECUTION FRAMEWORK

3.1 Symbolic Execution

In Algorithm 1, we show a symbolic execution algorithm called symExec, a general version of the one in [48]. symExec takes a program prog (compiled to low-level instructions such as LLVM IR) and a state selection strategy (strategy) as input, and generates a set of test cases (tests). symExec symbolically explores the branches of prog (different from [16] which explores one instruction each time) and stores the progress in a list of symbolic states (states). strategy is used to select a state from states for execution at each step. When the execution of one program path is finished, a test is generated and added to tests. Next, we describe symExec step by step.

At Line 2, we initialize tests and states to be empty. Then, we append prog’s initial state which represents the prog’s entry block to states (Line 3) and calls update to update strategy (Line 4). update is an auxiliary function that is called whenever a new state is added or a pending state is updated. We will discuss update later in this section. Next, at Line 5, the main loop for symbolic exploration starts. The loop terminates when states becomes empty, i.e., no available state can be explored, or the time limit has been reached. Inside the main loop, we first call selectState, another auxiliary function described later, for selecting a state (state) from states where each state represents a branch under exploration. We continuously execute the instructions of state symbolically (Line 8) and meanwhile check if the current execution violates any of the predefined security properties. The execution stops when we encounter an EXIT instruction, a security violation, or a FORK instruction (Line 7). When reaching an EXIT instruction that indicates the end of a program path or detecting a security violation, we call an external constraint solver to construct a new concrete test (Line 10) and remove state from states so that we stop further execution on state (Line 11). FORK instructions indicate a branching point, for which we need to copy state to create a forked state forked (Line 13). state and forked then represent the two new branches, respectively. We append forked to states, and update both state and forked (Line 14 to Line 16). After the main loop finally finishes, tests is returned.

Objective of symExec. Given an input program, the objective of running symExec with strategy is to find a set of concrete tests achieving the maximal coverage within a fixed amount of time:

$$\arg \max_{\text{tests} = \text{symExec} (\text{prog}, \text{strategy})} \frac{\bigcup_{t \in \text{tests}} \text{coverage}(t)}{\text{symExecTime}}$$

Equation 1, where coverage(t) measures the coverage of test t (we use line coverage in this work) and symExecTime is the time spent on running symExec. Achieving Equation 1 is challenging as the number of pending states is exponential to the number of forks and it is hard to predict what tests and the coverage of the tests a selected state will result in. The key is to construct a state selection strategy that can select the most promising states leading to high-quality tests.
3.2 State Selection Strategy

Next, we formally define a state selection strategy.

Definition 1 (State selection strategy). A state selection strategy is a mapping from symbolic states to real value scores that measure the importance of the states for exploration. To apply a state selection strategy in symExec, we need two auxiliary functions described at a high level below:

- `selectState`: The inputs of `selectState` are a list of pending symbolic states `states` and a state selection strategy `strategy`. Each time invoked (Line 6 of Algorithm 1), `selectState` leverages the importance scores returned by `strategy` to select a state from `states` for the next exploration step. `selectState` can be deterministic or probabilistic, e.g., normalizing the scores into a probability distribution and drawing a sample from the distribution.
- `update`: When a new pending state is added (Lines 4 and 16 of Algorithm 1) or a currently pending state enters a new branch (Line 15 of Algorithm 1), `update` is called to update the internal mechanics of `strategy` for computing the importance scores and also the scores themselves.

The detailed implementation of `selectState` and `update` depends on each specific strategy. Next, we provide the depth-first search (DFS) strategy in KLEE [16] as an example.

Example 1. The DFS strategy always selects the state representing the deepest path before exploring other paths.

- `strategy`: maps each pending state to its depth, i.e., the number of forks executed for the path that the state explores.
- `selectState`: selects the state with the largest depth.
- `update`: updates the depth of the input state.

Algorithm 1: Branch-based symbolic execution execution

```
Algorithm 1: Branch-based symbolic execution

Procedure symExec(prog, strategy)
  Input : prog, an input program.
  strategy, a state selection strategy.
  Output: tests, a set of generated test cases.
  tests ← emptySet(); states ← emptyList()
  states.append(prog.initialState)
  update(prog.initialState, strategy)
  while states.size > 0 and !TIMEOUT do
    state ← selectState(states, strategy)
    while state.inst ≠ EXIT and !state.foundViolation and
        state.inst ≠ FORK do
      executeInstruction(state)
      if state.inst = EXIT or state.foundViolation then
        tests.add(generateTest(state))
        states.remove(state)
      else
        forked ← doFork(state)
        states.append(forked)
        update(state, strategy)
      end
    end
  end
  return tests
```

Algorithm 2: LEARCH’s update function

```
Algorithm 2: LEARCH’s update function

Procedure update(state, strategy)
  Input: state, the state to update.
  strategy, the LEARCH strategy.
  state.feature ← extractFeature(state)
  reward ← strategy.predict(state.feature)
  strategy.setReward(state, reward)
```

Ideal objective of a state selection strategy. In symExec, a selected state can produce different new states and finally different tests, depending on the program logic and subsequent selection decisions. Ideally, at selectState, we would want to consider the overall effect of each pending state (i.e., the states and tests produced from the state) and select states leading to tests that not only achieve higher coverage but also cost less time to obtain, such that symExec’s objective in Equation 1 is achieved. This criterion can be summarized in the reward function defined as:

\[
reward(state) = \frac{\sum_{d \in statesFrom(state) \setminus stateTime(d)} \text{coverage}(t)}{|\{t \in testsFrom(state)\}|}
\]

where `testsFrom(state)` and `statesFrom(state)` return the set of tests and the set of symbolic states originating from `state`, respectively. `stateTime(d)` returns the time spent on state `d`, including execution time, constraint solving time, etc. Intuitively, `reward` measures `state` by the total amount of coverage achieved by the tests involving `state` divided by the total amount of time `symExec` spends on `state` and the states produced from `state`. Then, an ideal strategy would always select `state` with the highest reward.

However, it is hard to exactly compute `reward` at each `selectState` step, because the states and the tests produced from `state` depend on future selections that are unknown at the current step. That is, we usually cannot calculate `testsFrom(state)` and `statesFrom(state)` ahead of time before `symExec` finishes. Due to this limitation, existing heuristics [16, 48] typically compute importance scores for states based on a certain manually designed property, as a proxy for `reward`. As a result, they often get stuck at certain program parts and cannot achieve high coverage.

3.3 A Learned State Selection Strategy: LEARCH

Now we introduce the LEARCH strategy. The core component of LEARCH is a machine learning regression model \( \phi: \mathbb{R}^n \rightarrow \mathbb{R} \) learned to estimate the reward in Equation 2 for each pending state. To achieve this, LEARCH extracts a vector of \( n \) features for the input state with a function called `extractFeature` and invokes \( \phi \) on the \( n \)-dimensional features. The choices of the features and \( \phi \) are discussed in Section 5.1.

The `selectState` function of LEARCH greedily selects the state with the highest estimated reward, i.e.:

\[
state = \arg \max_{s \in states} strategy.getReward(s)
\]

We also considered probabilistic sampling but found that the greedy one performed better. LEARCH’s `update` function is presented in Algorithm 2. At Line 2, we call `extractFeature` to extract the features for the input state. At Line 3, we leverage \( \phi \) to predict a reward for the state. Then at Line 4, the learned strategy updates the reward
of the state. Note that the expensive feature extraction and reward prediction are done once per update. In selectState, we only read the predicted rewards, avoiding unnecessary re-computations.

**Benefits of a learned strategy.** Different from existing heuristics [16, 48], the LEARCH strategy makes decisions based on multiple high-level features (including the ones from the heuristics) and directly estimates Equation 2 to optimize for Equation 1. Therefore, LEARCH can effectively explore the input program and rarely gets stuck. As a result, LEARCH achieves higher coverage and detects more security violations than manually designed heuristics.

## 4 LEARNING STATE SELECTION STRATEGIES

While a learned strategy can be effective, it is non-obvious how to learn the desired strategy. This is because a supervised dataset consisting of explored states and their ground-truth reward for training the machine learning model φ is not explicitly available. Next, we introduce techniques for extracting such a supervised dataset from the tests generated by `symExec`, from which φ can be obtained with off-the-shelf learning algorithms.

### 4.1 Assigning a Reward to Explored States

Given a set of training programs, we run `symExec` to obtain a set of tests where each test consists of a list of explored states and covers certain code. From the tests, we construct a novel representation of the tests, called *tests trees*, whose nodes are the explored states, and leverage the trees to calculate a reward for the explored states. Note that the calculation of Equation 2 is feasible during training because `symExec` has already finished for the training programs. Finally, a supervised dataset is built for training φ. Next, we describe how to achieve this step by step.

First, we formally define tests in our context.

**Definition 2 (Test).** A test generated by `symExec` for an input program is a tuple \((\text{states}, \text{input}, \text{cov})\). \text{states} is a list of symbolic states \([\text{state}_0, \text{state}_1, ..., \text{state}_n]\) selected by selectState at Line 6 of Algorithm 1. Each state represents a explored branch and the branch represented by \(\text{state}_j\) is followed by the branch represented by \(\text{state}_{j+1}\) in the control flow graph of the program. Therefore, \text{states} indicates the program path induced by test. \text{input} is a concrete input for the input program and is constructed by solving the path constraints of \(\text{state}_0\) with a constraint solver. The concrete execution of \text{input} follows the path indicated by \text{states} and achieves coverage \text{cov}.

**Example 2.** In Figure 4(a), we show the control flow graph (CFG) of an example program. The CFG consists of seven basic blocks and seven edges, and the edges between nodes \(f\) and \(c\) represent a loop. We symbolically execute the example program and generate three tests shown in Figure 4(b). Test 1 and 2 execute the loop once and twice, respectively, both covering block \(a, c, f\), and \(g\) (i.e., the results of the coverage function in Equations 1 and 2). Test 3 does not execute the loop but explores states \(a_0, b_0\), and \(d_0\), covering blocks \(a, b, c, f, g\), and \(d\). Note that for the examples, we show basic block coverage for simplicity. In our implementation, we used line coverage. We record the time spent by `symExec` on each state (i.e., the results of the stateTime function in Equation 2) in Figure 4(c).

After generating tests for a training program, we construct a tests tree defined in the following.

**Definition 3 (Tests tree).** Given a series of tests \([\text{test}^0, \text{test}^1, ..., \text{test}^m]\) for a program, we construct a tests tree whose nodes are the explored states of the tests (i.e., those in the \text{states} field). For each \text{test}^j, we go over all pairs of states \(\text{state}_j^i\) and \(\text{state}_j^{i+1}\) and set \(\text{state}_j^i\) as the parent of \(\text{state}_j^{i+1}\) in the tree.

**Example 3.** In Figure 4(d), we show a tests tree constructed from the tests in Figure 4(b). Each tree root from the root to a leaf corresponds to the explored states of a test. For example, the leftmost path \(a_0 \rightarrow c_0 \rightarrow f_0 \rightarrow g_0\) consists of the states of test 1. At the left-hand side of each leaf node, we annotate the number of new blocks covered by the corresponding test. For instance, test 1 covers four new blocks: \(a, c, f, g\). Then, test 2 does not yield new coverage because the four blocks covered by test 2 were already covered by test 1 before. Test 3 covers two new blocks: \(b\) and \(d\).

The tests tree representation recovers the hierarchy of the explored states and provides a structure for conveniently calculating \text{testsFrom(state)} and \text{statesFrom(state)} by considering the descendants and the paths of each explored state, respectively. As a result, the reward can be efficiently computed.

**Calculate a reward for explored states.** To calculate a reward (Equation 2) for each \text{state}, we need to calculate the numerator, i.e., the total coverage achieved by all tests involving \text{state}, and the denominator, i.e., the total amount of time spent by \text{state} and its descendants. We compute those information with the tests trees in a bottom-up recursive fashion.

To compute the numerator \text{totalCov} for each \text{state}, we compute the coverage achieved by the tests involving \text{state}. This is equal
to summing up the new coverage (newCov) of all the leaves that are descendants of state in the tests trees and can be done in a recursive way as follows:

$$\text{totalCov(state)} = \begin{cases} 
\text{newCov(state)} & \text{if state is a leaf}, \\
\sum_{c \in \text{children}(state)} \text{totalCov}(c) & \text{otherwise}.
\end{cases}$$

To compute the denominator totalTime, we sum up the time spent on the considered state and its descendants via the following recursive equation:

$$\text{totalTime(state)} = \text{stateTime(state)} + \sum_{c \in \text{children}(state)} \text{totalTime}(c)$$

Then the reward can be computed by $\text{reward(state)} = \frac{\text{totalCov(state)}}{\text{totalTime(state)}}$.

**Example 4.** For each state in Figure 4(d), we compute totalTime, totalCov, and reward in Figure 4(e).

### 4.2 Strategy Learning Algorithms

We now present the final algorithms for learning LEARCH.

**Generate a supervised dataset.** In Algorithm 3, we present a procedure named `genData` for generating a supervised dataset. The inputs of `genData` are a set of training programs `progs` and a set of state selection strategies. First, at Line 2, we initialize the supervised dataset (dataset) to an empty set. Then, for each strategy in strategies and each program in progs (the loops from Line 3 to Line 7), we run `symExec` to generate a set of tests `tests` (Line 5). Next, at Line 6, a new supervised dataset is extracted from the tests with the techniques described in Section 4.1. The new dataset is added to `dataset` (Line 7). After the loops finish, `dataset` is returned.

**Iterative learning for producing multiple learned strategies.** While a single learned strategy is already more effective than existing heuristics [16, 48], we found that using multiple models during inference time can improve the tests generated by `symExec` even more (a form of ensemble learning). This is because the space of symbolic states is exponentially large and multiple strategies can explore a more diverse set of states than a single strategy. We propose an iterative algorithm called `iterLearn` in Algorithm 4 that trains multiple strategies. To incorporate the knowledge of existing heuristics into LEARCH, we treat them as an input (strategies) to `iterLearn` and leverage them in the data generation process.

**Algorithm 3: Generating a supervised dataset**

```plaintext
Procedure genData(progs, strategies)
    Input: progs, a set of training programs.
           strategies, a set of state selection strategies.
    Output: dataset, a supervised dataset.
    dataset ← emptySet()
    for strategy in strategies do
        for prog in progs do
            tests ← symExec(prog, strategy)
            newData ← dataFromTests(tests)
            dataset ← dataset ∪ newData
    return dataset
```

**Algorithm 4: Iterative learning**

```plaintext
Procedure iterLearn(progs, strategies, N)
    Input: progs, a set of training programs.
           strategies, a set of manual heuristics.
           N, the number of training iterations.
    Output: learned, a set of learned strategies.
    dataset ← emptySet(); learned ← emptySet()
    for i ← 1 to N do
        newData ← genData(progs, strategies)
        dataset ← dataset ∪ newData
        newStrategy ← trainStrategy(dataset)
        learned.add(newStrategy)
        strategies ← {newStrategy}
    return learned
```

`iterLearn` first initializes a supervised dataset `dataset` and a set of learned strategies `learned` to empty sets (Line 2). Then it starts a loop from Line 3 to Line 8 with N iterations. For each iteration, `genData` (Algorithm 3) is called to generate new supervised data using `strategies` (Line 4) and the new data is added to `dataset` (Line 5). Then, a new strategy `newStrategy` is trained at Line 6. To achieve this, we run an off-the-shelf learning algorithm on `dataset`, represented by the `trainStrategy` function. Then, `newStrategy` is added to `learned` (Line 7). At Line 8, we assign `newStrategy` as the only element in `strategies`. This indicates that `genData` is called with the manual heuristics only at the first loop iteration. After the first loop iteration, the learned strategy obtained from the previous iteration is used to generate new supervised data. After the loop finishes, we return the N learned strategies.

Note that our learning pipeline is general and can be extended to optimizing for other objectives (e.g., detecting heap errors) just by choosing an appropriate reward function (e.g., the number of heap access visited during execution) and designing indicative features. Moreover, LEARCH employs an offline learning scheme, i.e., training is done beforehand and the learned strategies are not modified at inference time. One natural future work item is to extend LEARCH with online learning where we utilize already explored states of the input program to improve the strategies at inference time. Online learning can help LEARCH generalize better to programs that are drastically different from the training programs.

## 5 INSTANTIATING LEARCH ON KLEE

In this section, we describe how to instantiate LEARCH in KLEE [16]. For more implementation details, please refer to LEARCH’s open source repository at https://github.com/eth-sri/learch. While LEARCH is general and can be applied to other symbolic execution frameworks, we chose KLEE because it is widely adopted in many applications, including hybrid fuzzing [27] and others [23, 24, 36, 53, 76]. We believe the benefits of LEARCH can be quickly transferred to the downstream applications and other systems [8, 22, 52, 68, 74].

### 5.1 Features and Model

We describe the features and the machine learning model of LEARCH.
To extract the bag-of-word representation, we go over each path that has been explored before. The fixed length of the subpaths is a hyper-parameter that is represented by an expression tree and traverse the tree to obtain the count of each node type.

The last five statistics are borrowed from existing expert-designed heuristics [16, 48]. By including them as features of LEARCH, we enable LEARCH to learn the advantages of those heuristics. Feature depth calculates the number of forks that happened along state’s path. Feature cpicnt records the number of instructions executed inside state’s current function. Feature icnt is the number of times for which the current instruction of state is executed. Feature covNew records the last newly covered instruction for state and calculates its distance to the current instruction. For feature subpath, we track the subpaths of state (i.e., the last branches visited by state’s path [48]) and return the number of times for which the subpaths have been explored before. The fixed length of the subpaths is a hyper-parameter and we used 1, 2, 4, and 8, as done in [48].

Model selection. LEARCH requires a machine learning model that transforms the features described above to an estimated reward (Line 3 of Algorithm 2). Any regression model can be adopted in our setting. We leverage the feedforward neural network model as it yielded good results in practice. We also tried simpler linear regression and more complicated recurrent neural networks, but found that feedforward networks achieved the best results. More results on model selection are discussed in Section 6.6.

Leverage multiple learned strategies. As described in Section 4.2, we apply ensemble learning to train N models and construct N strategies. To apply the N strategies during inference time, we simply divide the total time budget into N equal slots, run KLEE with each learned strategy independently on one slot, and union the tests from all the runs. This results in tests that achieve higher coverage and detect more security violations than using a single strategy for the full time budget. This is because different learned strategies can explore an even more diverse set of program parts. Moreover, KLEE usually generates tests quickly in the beginning and saturates later. One time slot is usually already enough for a learned strategy to generate a reasonably good set of tests.

5.2 Security Violations

An important aspect of symbolic execution tools is to detect violations of program properties that can lead to security issues. The original KLEE detects certain types of errors. However, we found that it usually only reports failures of the symbolic execution model such as errors of the symbolic memory model, reference of external objects, and unhandled instructions. Those errors are usually not triggered concretely and do not lead to security violations.

In this work, we leverage Clang’s Undefined Behavior Sanitizer (UBSan) [4] to instrument the input program and label five kinds of security violations, listed below:

- **Integer overflow**: checks if the arithmetic operations overflow. The operations include addition, subtraction, multiplication, division, modulo, and negation of signed and unsigned integers.
- **Oversized shift**: checks if the amount shifted is equal to or greater than the bit-width of the variable shifted, or is less than zero.
- **Out-of-bounds array reads/writes**: checks if the indices of array reads and writes are equal to or greater than the array size.
- **Pointer overflow**: checks if pointer arithmetic overflows.
- **Null dereference**: detects the use of null pointers or creation of null dereferences.

When any UBSan violation happens, a specific function is called. We added handlers for capturing those functions and generating a concrete test case triggering the violations, except for integer overflows which has already been supported by KLEE. As a result, KLEE is able to generate reproducible concrete tests for the above violations. We note that KLEE is not restricted to UBSan violations but the difficulty of supporting more violations depends on the implementation of the handlers. For example, it would take a significant amount of effort to support the AddressSanitizer [63] in KLEE so we consider it as a future work item.

6 EXPERIMENTAL EVALUATION

We present an extensive evaluation of LEARCH aiming to answer the following questions:

- Can LEARCH cover more code than existing manual heuristics?
- Can LEARCH discover more security violations?
- Can LEARCH generate better initial seeds for fuzzing?
- What is the impact of LEARCH’s design choices?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>Size of state’s current call stack.</td>
</tr>
<tr>
<td>successor</td>
<td>Number of successors of state’s current basic block.</td>
</tr>
<tr>
<td>testCase</td>
<td>Number of test cases generated so far.</td>
</tr>
<tr>
<td>coverage</td>
<td>(1) Number of new instructions covered by state’s branch. (2) Number of new instructions covered along state’s path. (3) Number of new source lines covered by state’s branch. (4) Number of new source lines covered along state’s path.</td>
</tr>
<tr>
<td>constraint</td>
<td>Bag-of-word representation of state’s path constraints.</td>
</tr>
<tr>
<td>depth</td>
<td>Number of forks already performed along state’s path.</td>
</tr>
<tr>
<td>cpicnt</td>
<td>Number of instructions visited in state’s current function.</td>
</tr>
<tr>
<td>icnt</td>
<td>Number of times for which state’s current instruction has been visited.</td>
</tr>
<tr>
<td>covNew</td>
<td>Number of instructions executed by state since the last time a new instruction is covered.</td>
</tr>
<tr>
<td>subpath</td>
<td>Number of times for which state’s subpaths have been visited. The length of the subpaths can be 1, 2, 4, or 8.</td>
</tr>
</tbody>
</table>

Table 1: Features for representing a symbolic state state.
Table 2: The statistics of the programs used as test sets. MainLOC represents the main program lines without neither internal nor external library code, i.e., the lines of the source file containing the main function. ELOC represents the total executable lines in the final executable after KLEE’s optimizations, including internal libraries within the package but excluding external library code that KLEE automatically links. The numbers for coreutils were averaged from all 52 test programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Input format</th>
<th>Binary size</th>
<th>MainLOC</th>
<th>ELOC</th>
<th>KLEE settings for symbolic inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>coreutils</td>
<td>8.31</td>
<td>various</td>
<td>142 KB</td>
<td>330</td>
<td>1208</td>
<td>-sym-args 0 1 10 -sym-args 0 2 2 -sym-files 1 8 -sym-stdin 8</td>
</tr>
<tr>
<td>diff</td>
<td>3.7</td>
<td>cmd + text</td>
<td>548 KB</td>
<td>552</td>
<td>7,739</td>
<td>-sym-args 0 2 2 A B -sym-files 2 50</td>
</tr>
<tr>
<td>find</td>
<td>4.7.0</td>
<td>cmd + text</td>
<td>802 KB</td>
<td>256</td>
<td>11,472</td>
<td>-sym-args 0 3 10 -sym-files 1 40 -sym-stdin 40</td>
</tr>
<tr>
<td>grep</td>
<td>3.6</td>
<td>cmd + text</td>
<td>587 KB</td>
<td>1,167</td>
<td>9,545</td>
<td>-sym-args 0 2 2 -sym-arg 10 A -sym-files 1 50</td>
</tr>
<tr>
<td>gawk</td>
<td>5.1.0</td>
<td>awk + text</td>
<td>1.3 MB</td>
<td>604</td>
<td>24,079</td>
<td>-f A B -sym-files 2 50</td>
</tr>
<tr>
<td>patch</td>
<td>2.7.6</td>
<td>cmd + text + diff</td>
<td>466 KB</td>
<td>984</td>
<td>7,007</td>
<td>-sym-args 0 2 2 A B -sym-files 2 50</td>
</tr>
<tr>
<td>objcopy</td>
<td>2.36</td>
<td>cmd + elf</td>
<td>4.9 MB</td>
<td>2,513</td>
<td>48,895</td>
<td>-sym-args 0 2 2 A -sym-files 1 100</td>
</tr>
<tr>
<td>readelf</td>
<td>2.36</td>
<td>elf</td>
<td>2.4 MB</td>
<td>10,381</td>
<td>28,522</td>
<td>-a A -sym-files 1 100</td>
</tr>
<tr>
<td>make</td>
<td>4.3</td>
<td>Makefile</td>
<td>466 KB</td>
<td>888</td>
<td>7,862</td>
<td>-n -f A -sym-files 1 40</td>
</tr>
<tr>
<td>cjson</td>
<td>1.7.14</td>
<td>json</td>
<td>83 KB</td>
<td>71</td>
<td>610</td>
<td>A -sym-files 1 100 yes</td>
</tr>
<tr>
<td>sqlite</td>
<td>3.33.0</td>
<td>sql commands</td>
<td>2.1 MB</td>
<td>35,691</td>
<td>46,388</td>
<td>-sym-stdin 20</td>
</tr>
</tbody>
</table>

6.1 Evaluation Setup

Now we describe the setup for our experimental evaluation.

Benchmarks. We evaluated LEARCh on coreutils (version 8.31) and 10 real-world programs (listed in Table 2). coreutils is a standard benchmark for evaluating symbolic execution techniques [16, 22, 48, 50]. We excluded 3 coreutils utilities (kill, ptx, and yes) that caused non-deterministic behaviors in our initial experiments. As a result, we used 103 coreutils programs in our evaluation. The 10 real-world programs are much larger than most coreutils programs, deal with various input formats, and are widely used in fuzzing and symbolic execution literature [7, 12, 15, 21, 43, 72].

We randomly selected 51 of the 103 coreutils programs for training LEARCh. The rest 52 coreutils programs and the 10 real-world programs were used as test sets for evaluating LEARCh’s performance on unseen programs. The statistics of both test sets can be found in Table 2. The coreutils test set has overlapping code with the training set as they are from the same package [5]. This represents a common and valid use case where developers train LEARCh on programs from their code base and then run it to test other programs from the same code base. Note that our use case is different from other security tasks based on machine learning such as binary function recognition [5, 10, 67] where sharing of code between train-test splits must be avoided as training on target binaries is impossible due to unavailable source code. The 10 real-world programs are from packages different from coreutils and thus share less code with coreutils. They are used to demonstrate that LEARCh generalizes well across different code bases. That is, once trained (e.g., with coreutils in our evaluation), LEARCh can directly be used to test other software packages.

Baselines. We adopted existing manually crafted heuristics created for KLEE as baselines [16, 48]. We do not compare with [74] because it is not part of KLEE and we did not find its implementation available. We ran all KLEE’s individual heuristics on our coreutils test set and only present the top four due to space limit:

- **rss** (random state search): each time selects a state uniformly at random from the list of pending states.
- **rps** (random path search): constructs a binary execution tree where the leaves are the pending states and the internal nodes are explored states that produce the pending states. To select a state, rps traverses the tree in a top-down fashion, picks a child of internal nodes randomly until reaching a leaf, and returns the reached leaf as the selection result. The leaves closer to the root are more likely to be selected.
- **nurs:cpicnt** and **nurs:depth**: both are instances of the **nurs** (non-uniform random search) family. They sample a state from a distribution where the probability of each state is heuristically defined by cpicnt and depth, respectively. See Table 1 and Section 5.1 for the definitions of cpicnt and depth.

We also compare LEARCh with combinations of multiple heuristics:

- **sgs** (subpath-guided search) [48]: selects a state whose subpath (defined in Table 1 and Section 5.1) was explored least often. To achieve the best results, the authors of [48] ran four independent instances of sgs where subpath lengths were configured to 1, 2, 4, and 8, respectively. Each instance spent a quarter of the total time limit, and then the resulted test cases were combined. We followed this in our evaluation.
- **portfolio**: a portfolio of four different heuristics: rps, nurs:cpicnt, nurs:depth, and sgs. Like sgs and LEARCh, we ran each heuristic of portfolio as an independent instance that spends a quarter of the total time budget.

Different from Algorithm 1 which selects a state per branch, the original KLEE performs state selection per instruction. In our initial experiments on coreutils, we found that running the heuristics with Algorithm 1 gave better results and thus used Algorithm 1 for all heuristics in our evaluation. This means that our baselines are already stronger than their counterparts in the original KLEE.

KLEE settings. KLEE provides users with options to specify the number and length of symbolic inputs (e.g., command-line arguments, files, stdin, and stdout) to an input program. The symbolic options we used are listed in Table 2. We followed prior works [16, 48] to set symbolic inputs for coreutils programs. For the 10 real-world programs, we configured the symbolic options based on their input formats and prior works [15, 43].
When evaluating the `coreutils` test set, we set the time limit to 1h for each search strategy, following [16, 48]. For the real-world programs, the time limit was 8h. The memory limit for all programs was 4GB, which is higher than KLEE’s default budget (2GB) and the limit used by prior works [15, 21]. We did not input any initial seed test to KLEE. Whenever necessary, we repeated our experiments for 20 times and report the mean and standard deviation.

**Training and testing LEARCH.** We trained LEARCH by running Algorithm 4 on the 51 `coreutils` training programs. The set of initial strategies consisted of `rps`, `nurs:cpicnt`, `nurs:depth`, and `sgs` (with subpath lengths 1, 2, and 4) as these strategies performed the best on the training programs. We ran Algorithm 4 for 4 iterations to train 4 strategies (feedforward networks with 3 linear layers, 64 hidden dimension, and ReLU activations). We did not include more trained strategies because more strategies did not significantly increase LEARCH’s performance. Each iteration spent around 4h (2h on `symExec`, 1h on `dataFromTests`, and 1h on `trainStrategy`). When running LEARCH on the test set, we ran each strategy for a quarter of the total time limit and combined the resulted test cases.

**Versions and platform.** We implemented LEARCH on KLEE 2.1 and LLVM 6.0. We used pytorch 1.4.0 for learning. All symbolic execution experiments were performed on a machine with 4 Intel Xeon E5-2690 v4 CPUs (2.60 GHz) and 512 GB RAM. Each KLEE instance was restricted to running on one core. The machine learning models were trained on a machine with RTX 2080 Ti GPUs.

### 6.2 Code Coverage

In this section, we present our evaluation on code coverage, i.e., line coverage measured with gcov [2]. We first report absolute line coverage for all files in the package. Then, we present and discuss the percentage of covered lines.

**Line coverage for `coreutils` programs.** In Figure 5(a), we present the coverage of each strategy on the 52 `coreutils` testing programs. On average, LEARCH (green bar) covered 618 lines for all files in the
package. The best individual heuristic was rps (purple bar), which covered 546 lines. That is, Learch achieved at least 13% more coverage than any individual heuristic. portfolio (purple bar) was the best combined heuristic but still covered 44 lines less than Learch.

In Figure 5(b), we plot the number of programs where each strategy achieved the best coverage among all the strategies. For ties, we count one for each strategy. For 29 programs, Learch was the best strategy, outperforming portfolio by 3 and other heuristics by a large margin. For the other 23 programs, at least one heuristic performed better than Learch, but usually only by a small margin. Moreover, we found that Learch gave more benefits on larger coreutils programs. For example, for the largest 5 programs, Learch achieved ~30% more coverage than portfolio, compared to ~8% overall for all 52 programs. For smaller programs, the manual heuristics already covered most parts of the programs so Learch hardly improved upon them.

Line coverage for real-world programs. In Figure 6, we plot the coverage of each strategy on the 10 real-world programs. To generate the coverage curve for combinations of strategies, we treat independent runs of each strategy sequentially. On average, Learch covered 2,433 lines while the best manual heuristic, portfolio, covered 2,023 lines. Overall, Learch outperformed all the manual heuristics by >20%.

Judging from the mean values, Learch was the best strategy for 8 of the 10 real-world programs, except for cjson and find. For cjson, Learch was the second best, covering 10 lines less than portfolio. For find, Learch covered 242 and 191 less lines than portfolio and rps, respectively. Learch’s superior performance in code coverage was consistent among the 10 programs while all manual heuristics were unstable. For example, portfolio did well on cjson and find but not on objcopy and sqlite. Similarly, sgs performed well on gawk but poorly on diff.

For sqlite, the standard deviations of all strategies were high. The reason is that, during some runs, the strategies were unable to generate a few important test cases, resulting in thousand of lines less coverage than other runs. This happened the least often for Learch so Learch’s mean coverage was the highest.

Percentage of covered lines. Apart from absolute line coverage, we calculated the percentage of covered lines to investigate how thoroughly the test programs are covered by KLEE and the strategies. We measured the percentage of covered MainLOCs and ELOCs (mean value from 20 runs). The total number of MainLOCs and ELOCs for each test program can be found in Table 2.

MainLOC was used for measuring coverage in [16, 48] and refers to the main program lines, i.e., the lines of the source file containing the main function. It does not include internal and external library code that can be invoked by multiple programs to avoid counting them multiple times (see [16] for details on the advantages of using MainLOC). The results for MainLOC percentages are presented in Table 3. Learch achieved the highest coverage for most cases. We can observe that the percentages for all strategies decrease with increasing program size: for small programs such as coreutils, cjson and find, all strategies achieved relatively high coverage (e.g., >60%); while for large programs such as readelf and sqlite, KLEE only covered ~10% MainLOCs.

ELOC, short for executable lines of code, represents the total executable lines in the final executable after KLEE’s optimizations. In [16, 48], it was used for measuring program size and included external library code that KLEE automatically links. In our work, we use ELOC for measuring coverage and thus excluded external library code that does not belong to the program package. The internal library code from the package was included. The results on the percentage of covered ELOCs are presented in Table 4. Learch still covered most portions of code for most cases. However, even with the best strategy, KLEE covered only a very small portion of code (e.g., <10% for the real-world programs).

Our results on percentage of covered lines show that it is still challenging to scale symbolic executor like KLEE to large programs. While Learch improves on other search strategies, efforts on other directions are also needed, such as reducing memory consumption [15, 21], accelerating constraint solving [9, 28, 32, 61], and executing program fragments instead of the whole program [59, 70].

Summary on generalizability. Learch’s effectiveness in code coverage demonstrates its generalizability: it generalizes between programs from the same package (i.e., training and testing on coreutils). This is because these programs usually share the same code and our learning method can capture this. More interestingly, Learch also generalizes from a package of smaller programs (i.e., training on coreutils) to other packages whose code is different from the training package and programs are much larger (i.e., testing on the real-world programs). This is likely because Learch’s features capture the importance of symbolic states even for programs different from the training set.

<table>
<thead>
<tr>
<th>Table 3: Percentage of covered MainLOCs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rss</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>coreutils</td>
</tr>
<tr>
<td>diff</td>
</tr>
<tr>
<td>find</td>
</tr>
<tr>
<td>grep</td>
</tr>
<tr>
<td>gawk</td>
</tr>
<tr>
<td>patch</td>
</tr>
<tr>
<td>objcopy</td>
</tr>
<tr>
<td>readelf</td>
</tr>
<tr>
<td>make</td>
</tr>
<tr>
<td>cjson</td>
</tr>
<tr>
<td>sqlite</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4: Percentage of covered ELOCs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rss</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>coreutils</td>
</tr>
<tr>
<td>diff</td>
</tr>
<tr>
<td>find</td>
</tr>
<tr>
<td>grep</td>
</tr>
<tr>
<td>gawk</td>
</tr>
<tr>
<td>patch</td>
</tr>
<tr>
<td>objcopy</td>
</tr>
<tr>
<td>readelf</td>
</tr>
<tr>
<td>make</td>
</tr>
<tr>
<td>cjson</td>
</tr>
<tr>
<td>sqlite</td>
</tr>
</tbody>
</table>
6.3 Detecting Security Violations

We ran the strategies on program instrumented with UBSan checkers to evaluate their capability of detecting UBSan violations. All the detected violations are true positives because they can be triggered by the generated test cases.

In Figure 7, we present the number of UBSan violations detected by the strategies. For the coreutils test set (Figure 7(a)), LEARCH detects 88 violations in total, outperforming the manual heuristics by >12%. For the 10 real-world programs (Figure 7(b)), LEARCH outperformed all manual heuristics except for sgs which found the same number of violations as LEARCH, even though LEARCH achieved higher coverage than sgs. This is likely because the parts of the programs explored by sgs contained more UBSan labels. We provide a manual analysis of the UBSan violations detected with LEARCH in Section 6.5.

6.4 Seeding for Fuzzing

Fuzzing has gained substantial interest recently [1, 27, 30, 45, 72]. It is shown that fuzzing performance heavily depends on the choices of initial seeds [39, 45]. While the initial seeds used in prior works are usually empty, randomly generated, or manually constructed [12, 26, 45, 71], symbolic execution can be used to automatically generate fuzzing seeds (see [29] for how initial seeds generated by KLEE compare to simple and expert seeds). In this work, we investigate if LEARCH can generate better fuzzing seeds than the manual heuristics.

We selected AFL (version 2.52b) [1] due to its popularity and ran it on the four largest programs in our real-world benchmarks whose input format supports AFL-style fuzzing: objcopy, readelf, make, and sqlite. For each program and each strategy, we constructed the initial seed set by selecting the top three tests from our previous experiment (i.e., Figure 6) based on the best coverage and ran AFL starting from the initial seeds for 8h. We selected only three initial seeds because using a small set of initial seeds is recommended by AFL and adopted by many fuzzing works [3, 7, 27, 45, 51]. Aware of the randomness in AFL, we repeated each run for 20 times and report the mean and standard deviation.

Discovering paths. AFL generates a test when a new path is triggered. Therefore, one of the most direct indicator of AFL’s progress is path coverage, i.e., the number of discovered paths.
6.5 Manual Analysis of Security Violations

Unlike standard crash bugs, UBSan violations may not crash the program but can indicate deeper, functional misbehaviors. Moreover, some UBSan violations may be benign. In this section, we performed a manual inspection of the violations found with LEARCH to understand the severity of the violations.

The vast majority of detected violations were overflows. We manually inspected 112 violations detected by running KLEE with LEARCH, as well as 152 violations discovered by AFL with initial seeds from LEARCH. From these violations, we identified 46 potential bugs (see column “Bug reports” in Table 5). These potential bugs can result in logical errors (e.g., setting a wrong value, moving a file cursor to a wrong position, wrong control flow, etc.) or out-of-bound reads/writes. The remaining violations were benign cases, mainly categorized as: (1) the programs already consider the violation and have specialized handlers; (2) the operations are used for hashing or generating random numbers so the overflows do not affect program logic; (3) the related variables are set to a new value or not used after the violations.

We reported the 46 potential bugs to the developers and 13 of them were confirmed as true bugs. The other potential bugs were recognized as false positives by the developers. Among the confirmed bugs, 11 bugs have been fixed or will be fixed, or the developers are discussing how to fix them. The number of confirmed and fixed bugs for each program is listed in Table 5.

We investigate the usefulness of LEARCH’s design choices. Due to space limit, we mainly present the results on coreutils. For the real-world programs, we observed the same phenomenon as coreutils.

### 6.6 Effectiveness of Design Choices

We show four examples of UBSan violations detected with LEARCH. The first three are confirmed bugs and the last one is a benign case. In Figure 9, &p->pred_left is null pointer but the code tries to dereference it. This bug was detected by running KLEE with LEARCH on find. In Figure 10, the addition at Line 4 and the multiplication at Line 5 can overflow, affecting the value of pos and leading to a wrong file position when reading a binary executable file with objcopy. Figure 11 is taken from make and contains a subtract overflow at Line 5 which results in wrong or even out-of-bound array accesses. AFL detected Figures 10 and 11 using the initial seeds from LEARCH. Figure 12 shows a benign subtraction overflow detected by KLEE with LEARCH for the coreutils tool sort. The code computes the logarithm of a number stored in the unsigned char array tmpa. To decide if the current array element is a digit, the code uses the macro ISDIGIT which overflows when c < ‘0’. When this happens, ISDIGIT returns false, which is desired as c is not a digit. Therefore, the overflow case was already considered and handled.

#### Examples

We show four examples of UBSan violations detected with LEARCH. The first three are confirmed bugs and the last one is a benign case. In Figure 9, &p->pred_left is null pointer but the code tries to dereference it. This bug was detected by running KLEE with LEARCH on find. In Figure 10, the addition at Line 4 and the multiplication at Line 5 can overflow, affecting the value of pos and leading to a wrong file position when reading a binary executable file with objcopy. Figure 11 is taken from make and contains a subtract overflow at Line 5 which results in wrong or even out-of-bound array accesses. AFL detected Figures 10 and 11 using the initial seeds from LEARCH. Figure 12 shows a benign subtraction overflow detected by KLEE with LEARCH for the coreutils tool sort. The code computes the logarithm of a number stored in the unsigned char array tmpa. To decide if the current array element is a digit, the code uses the macro ISDIGIT which overflows when c < ‘0’. When this happens, ISDIGIT returns false, which is desired as c is not a digit. Therefore, the overflow case was already considered and handled.

#### 6.6 Effectiveness of Design Choices

We investigate the usefulness of LEARCH’s design choices. Due to space limit, we mainly present the results on coreutils. For the real-world programs, we observed the same phenomenon as coreutils.

### Performance of individual strategies

As described in Section 6.1, LEARCH consists of four learned strategies. We ran each strategy for 1h on the coreutils test set and compare the results with LEARCH (a union of the four strategies each running for 15m) in
7 RELATED WORK

We discuss works closely related to ours.

Symbolic execution. Symbolic execution based testing techniques have been developed for decades [18, 44], yielding a number of applications [23, 24, 27, 36, 53, 76] and systems [8, 16, 17, 22, 52, 68, 74]. The main challenges in symbolic execution include path explosion and expensive constraint solving [18]. A number of manual heuristics have been proposed for selecting promising paths [16, 48]. Our learning-based strategy LEARCh significantly outperforms those heuristics. Other orthogonal attempts for easing the path explosion problem include state merging [46], state pruning [13, 14, 21, 70], and code transformation [25]. A number of works focus on improving the performance of constraint solvers [9, 28, 32, 61]. Some works combine the constraint solving process with the symbolic execution framework by solving multiple path constraints once [77], leveraging pending path constraints [43], and introducing neural constraints [66]. While most of the above approaches aim to explore the whole program (same as our goal), directed symbolic execution aims to reach certain program parts or changes [50, 56, 73].

8 CONCLUSION

In this work, we introduced LEARCh, a learning-based state selection strategy for symbolic execution. LEARCh works by estimating a reward for each state and selecting the state with the highest reward to maximize coverage while minimizing time cost. We construct LEARCh by applying off-the-shelf regression learning on a supervised dataset extracted from the tests generated by running symbolic execution on a set of training programs. The training process is iterative and constructs multiple strategies which produce more diverse tests than a single strategy. LEARCh benefits from existing heuristics by incorporating them in both training and feature extraction.

We instantiated LEARCh on KLEE [16] and evaluated it on the coreutils programs and ten real-world programs. The results demonstrated that LEARCh is effective and can produce higher-quality tests than existing manually designed heuristics, either individual ones or combined as portfolios: LEARCh’s tests yielded more code coverage, detected more security violations, and were better candidates as initial seeds for fuzzers like AFL.