# 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

**ACM Reference Format:**
Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3484813


*CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea*

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484813

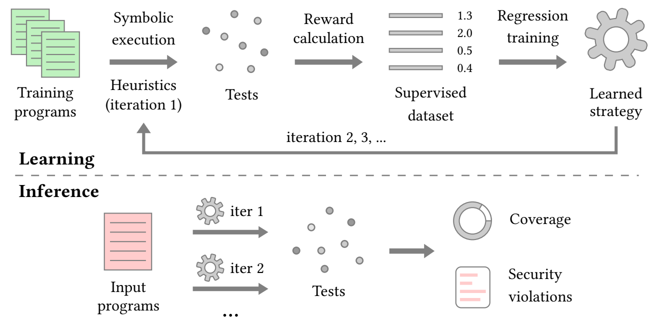



**Figure 1: Our recipe for learning a state selection strategy.**

## 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 search), 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[1] 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.

[1]Learch is publicly available at https://github.com/eth-sri/learch.

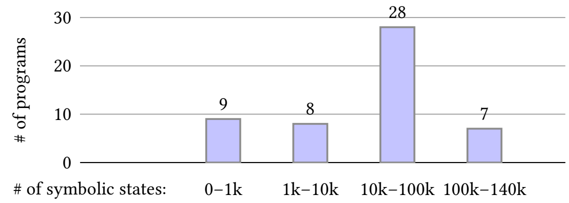



**Figure 2: Number of `coreutils` programs grouped by the average number of candidate states available at selection steps.**

**Main contributions.** Our main contributions are:

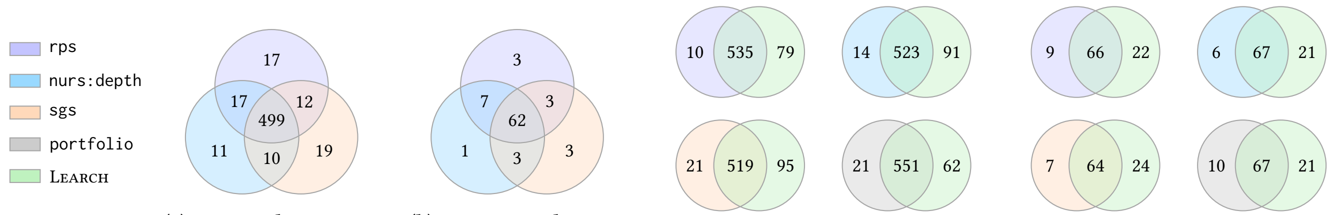
- Learch, a new learning-based state selection strategy for symbolic execution. (Section 3)
- A novel learning framework for constructing multiple learned strategies. (Section 4)
- A complete instantiation of Learch on the popular symbolic test generator KLEE. (Section 5)
- An extensive evaluation on 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 set 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.** Existing 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).




**(a) Average line coverage by heuristics.** **(b) UBSan violations found by heuristics.** **(c) Average line coverage: Learch v.s. heuristics.** **(d) UBSan violations: Learch v.s. heuristics.**

**Figure 3: Limitations of existing manually designed heuristics and how Learch outperforms them for our `coreutils` test set.**

**Opportunities for learning.** Based on the results in Figure 3 (a) and Figure 3 (b), we can compute the *union* of covered lines and detected UBSan violations for the three heuristics. The union achieved 585 line coverage and detected 82 UBSan violations, significantly higher than any individual heuristic. This indicates a promise of constructing an adaptive strategy that subsumes individual heuristics. In fact, the heuristics provide a precious knowledge base that facilitates learning such an adaptive strategy. Namely, the heuristics can be used to generate a diverse training dataset, capturing different selection behaviors, and the properties they rely on can be used as valuable features for a machine learning model. Another key advantage of learning is that while the reward calculation is impossible at inference time, it can be computed for the states explored at training time, from which we can obtain a direct estimator for the final coverage-guided objective. The above insights facilitate and motivate our learning scheme proposed in Figure 1.

To show evidence on the benefits of learning, we constructed another heuristic called `portfolio` by running the three heuristics, each for a third of the total time limit, and combined all produced test cases. We compare Learch with the four heuristics in Figure 3 (c) and Figure 3 (d). In terms of total coverage, Learch outperformed all four heuristics. Learch was able to cover most code covered by the heuristics, and exclusively covered more code than the heuristics. The same holds for the detection of UBSan violations.

**Scope and applicability.** Our work focuses on *purely symbolic execution* (i.e., no concrete execution happens during the test generation process), even though our idea may give hints for improving other approaches where it is tricky to select program branches for test generation, such as concolic testing [33, 57, 58, 62] and hybrid testing [27, 69, 75]. We aim to solve the *state selection* problem, while there are many orthogonal approaches for easing path explosion such as state merging [46] and state pruning [13, 14, 21, 70].

# 3 SYMBOLIC EXECUTION FRAMEWORK

In this section, we first present a general symbolic execution framework parameterized by a state selection strategy and then introduce the Learch state selection strategy.

## 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:

$$\arg\max_{tests=\texttt{symExec}(prog,strategy)} \frac{|\bigcup_{t \in tests} \texttt{coverage}(t)|}{\text{symExecTime}} \quad (1)$$

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.

**Algorithm 1:** Branch-based symbolic execution

```
1  Procedure symExec(prog, strategy)
     Input  : prog, an input program.
              strategy, a state selection strategy.
     Output : tests, a set of generated test cases.
2    tests ← emptySet();   states ← emptyList()
3    states.append(prog.initialState)
4    update(prog.initialState, strategy)
5    while states.size > 0 and !TIMEOUT do
6      state ← selectState(states, strategy)
7      while state.inst ≠ EXIT and !state.foundViolation and
         state.inst ≠ FORK do
8        executeInstruction(state)
9      if state.inst = EXIT or state.foundViolation then
10       tests.add(generateTest(state))
11       states.remove(state)
12     else                                  // state.inst = FORK
13       forked ← doFork(state)
14       states.append(forked)
15       update(state, strategy)
16       update(forked, strategy)
17   return 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 2:** Learch's `update` function

```
1  Procedure update(state, strategy)
     Input: state, the state to update.
            strategy, the Learch strategy.
2    state.feature ← extractFeature(state)
3    reward ← strategy.predict(state.feature)
4    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:

$$\texttt{reward}(state) = \frac{\left|\bigcup_{t \in \texttt{testsFrom}(state)} \texttt{coverage}(t)\right|}{\sum_{d \in \texttt{statesFrom}(state)} \texttt{stateTime}(d)} \tag{2}$$

where $\texttt{testsFrom}(state)$ and $\texttt{statesFrom}(state)$ return the set of tests and the set of symbolic states originating from *state*, respectively. $\texttt{stateTime}(d)$ 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 the 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 $\texttt{testsFrom}(state)$ and $\texttt{statesFrom}(state)$ 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.

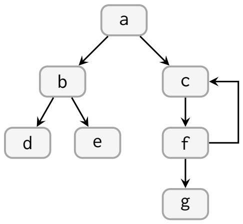
## 3.3 A Learned State Selection Strategy: Learch

Now we introduce the Learch strategy. The core component of Learch is a machine learning regression model $\varphi\colon \mathbb{R}^n \to \mathbb{R}$ 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 $\varphi$ on the $n$-dimensional features. The choices of the features and $\varphi$ are discussed in Section 5.1.

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

$$state = \arg\max_{s \in states} strategy.\text{getReward}(s).$$

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 $\varphi$ to predict a reward for the state. Then at Line 4, the learned strategy updates the reward



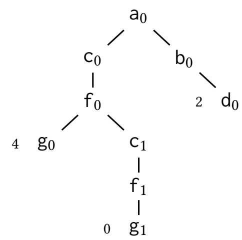

**(a) An example CFG.**

| | *states* | *cov* |
|---|---|---|
| 1 | $a_0$, $c_0$, $f_0$, $g_0$ | a, c, f, g |
| 2 | $a_0$, $c_0$, $f_0$, $c_1$, $f_1$, $g_1$ | a, c, f, g |
| 3 | $a_0$, $b_0$, $d_0$ | a, b, d |

**(b) Example tests generated by `symExec`.**

| $a_0$ | $c_0$ | $f_0$ | $g_0$ | $c_1$ | $f_1$ | $g_1$ | $b_0$ | $d_0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 1 | 1 | 2 | 2 | 2 |

**(c) Time spent by each state in seconds.**




**(d) A tests tree.**

| state | total_cov | total_time | reward |
|---|---|---|---|
| $a_0$ | 6 | 15 | 0.4 |
| $c_0$ | 4 | 10 | 0.4 |
| $f_0$ | 4 | 8 | 0.5 |
| $g_0$ | 4 | 2 | 2 |
| $c_1$ | 0 | 4 | 0 |
| $f_1$ | 0 | 3 | 0 |
| $g_1$ | 0 | 2 | 0 |
| $b_0$ | 2 | 4 | 0.5 |
| $d_0$ | 2 | 2 | 1 |

**(e) Reward for the states.**

**Figure 4: An example on assigning a reward to explored states of the tests generated by `symExec`.**

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 $\varphi$ is not explicitly available. Next, we introduce techniques for extracting such a supervised dataset from the tests generated by `symExec`, from which $\varphi$ 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 $\varphi$. 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 (*states*, *input*, *cov*). *states* is a list of symbolic states $[state_0, state_1, ..., state_n]$ selected by `selectState` at Line 6 of Algorithm 1. Each state represents a explored branch and the branch represented by $state_i$ is followed by the branch represented by $state_{i+1}$ in the control flow graph of the program. Therefore, *states* indicates the program path induced by *test*. *input* is a concrete input for the input program and is constructed by solving the path constraints of $state_n$ with a constraint solver. The concrete execution of *input* follows the path indicated by *states* and achieves coverage *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`, 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 $[test^0, test^1, ..., test^m]$ for a program, we construct a *tests tree* whose nodes are the explored states of the tests (i.e., those in the *states* field). For each $test^i$, we go over all pairs of states $state^i_j$ and $state^i_{j+1}$ and set $state^i_j$ as the parent of $state^i_{j+1}$ in the tree.

**Example 3.** In Figure 4(d), we show a tests tree constructed from the tests in Figure 4(b). Each tree path from the root to a leaf corresponds to the explored states of a test. For example, the leftmost path $a_0$–$c_0$–$f_0$–$g_0$ 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`, and `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 `testsFrom`(*state*) and `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 *state*, we need to calculate the numerator, i.e., the total coverage achieved by all tests involving *state*, and the denominator, i.e., the total amount of time spent by *state* and its descendants. We compute those information with the tests trees in a bottom-up recursive fashion.

To compute the numerator `totalCov` for each *state*, we compute the coverage achieved by the tests involving *state*. This is equal

**Algorithm 3:** Generating a supervised dataset

```
1 Procedure genData(progs, strategies)
      Input   : progs, a set of training programs.
                strategies, a set of state selection strategies.
      Output  : dataset, a supervised dataset.
2     dataset ← emptySet()
3     for strategy in strategies do
4         for prog in progs do
5             tests ← symExec(prog, strategy)
6             newData ← dataFromTests(tests)
7             dataset ← dataset ∪ newData
8     return dataset
```

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:

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

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

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

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

**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 4:** Iterative learning

```
1 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.
2     dataset ← emptySet();   learned ← emptySet()
3     for i ← 1 to N do
4         newData ← genData(progs, strategies)
5         dataset ← dataset ∪ newData
6         newStrategy ← trainStrategy(dataset)
7         learned.add(newStrategy)
8         strategies ← {newStrategy}
9     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.

**Table 1: Features for representing a symbolic state *state*.**

| Feature | Description |
|---|---|
| `stack` | Size of *state*'s current call stack. |
| `successor` | Number of successors of *state*'s current basic block. |
| `testCase` | Number of test cases generated so far. |
| `coverage` | (1) Number of new instructions covered by *state*'s branch.<br>(2) Number of new instructions covered along *state*'s path.<br>(3) Number of new source lines covered by *state*'s branch.<br>(4) Number of new source lines covered along *state*'s path. |
| `constraint` | Bag-of-word representation of *state*'s path constraints. |
| `depth` | Number of forks already performed along *state*'s path. |
| `cpicnt` | Number of instructions visited in *state*'s current function. |
| `icnt` | Number of times for which *state*'s current instruction has been visited. |
| `covNew` | Number of instructions executed by *state* since the last time a new instruction is covered. |
| `subpath` | Number of times for which *state*'s subpaths [48] have been visited. The length of the subpaths can be 1, 2, 4, or 8. |

**Features.** In Table 1, we list the features extracted for a symbolic state (*state*) by `extractFeature` at Line 2 of Algorithm 2. Feature `stack` calculates the size of *state*'s current call stack. The larger the call stack size, the deeper the execution goes into. Feature `successor` calculates the number of successors that *state*'s current basic block has. The more successors, the more paths the state can lead to. The next two features capture the execution progress. Feature `testCase` returns the number of already generated test cases. The `coverage` feature tracks the new instruction and line coverage achieved by *state*'s latest branch and the program path already explored by *state*, respectively. The states with more new coverage should be explored first. Feature `constraint` is a 32-dimensional vector containing the bag-of-word representation of the path constraints of *state*. To extract the bag-of-word representation, we go over each path constraint 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 hyperparameter 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 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.**

| Program | Version | Input format | Binary size | MainLOC | ELOC | KLEE settings for symbolic inputs |
|---|---|---|---|---|---|---|
| `coreutils` | 8.31 | various | 142 KB | 330 | 1208 | `-sym-args 0 1 10 -sym-args 0 2 2 -sym-files 1 8 -sym-stdin 8` |
| `diff` | 3.7 | cmd + text | 548 KB | 552 | 7,739 | `-sym-args 0 2 2 A B -sym-files 2 50` |
| `find` | 4.7.0 | cmd + text | 802 KB | 256 | 11,472 | `-sym-args 0 3 10 -sym-files 1 40 -sym-stdin 40` |
| `grep` | 3.6 | cmd + text | 587 KB | 1,167 | 9,545 | `-sym-args 0 2 2 -sym-arg 10 A -sym-files 1 50` |
| `gawk` | 5.1.0 | awk + text | 1.3 MB | 604 | 24,079 | `-f A B -sym-files 2 50` |
| `patch` | 2.7.6 | cmd + text + diff | 466 KB | 984 | 7,007 | `-sym-args 0 2 2 A B -sym-files 2 50` |
| `objcopy` | 2.36 | cmd + elf | 4.9 MB | 2,513 | 48,895 | `-sym-args 0 2 2 A -sym-files 1 100` |
| `readelf` | 2.36 | elf | 2.4 MB | 10,381 | 28,522 | `-a A -sym-files 1 100` |
| `make` | 4.3 | Makefile | 466 KB | 883 | 7,862 | `-n -f A -sym-files 1 40` |
| `cjson` | 1.7.14 | json | 83 KB | 71 | 610 | `A -sym-files 1 100 yes` |
| `sqlite` | 3.33.0 | sql commands | 2.1 MB | 35,691 | 46,388 | `-sym-stdin 20` |

## 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:

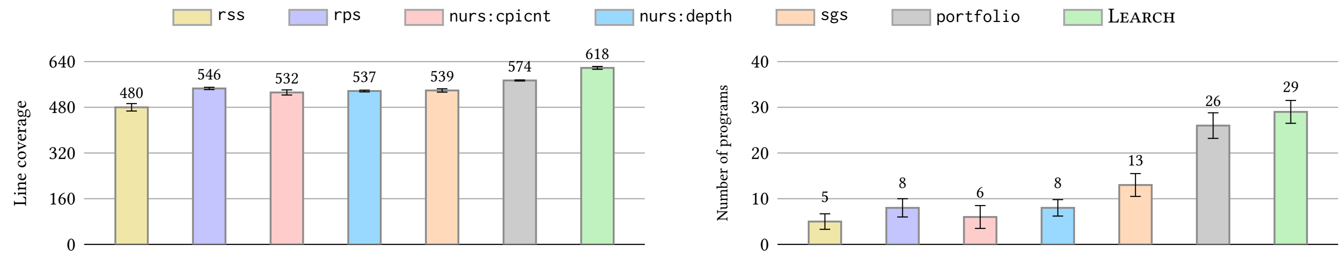
- `rss` (random state search): each time selects a state uniformly at random from the list of pending states.
- `rps` (random path search): constructs a binary execution tree where the leaves are the pending states and the internal nodes are explored states that produce the pending states. To select 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.

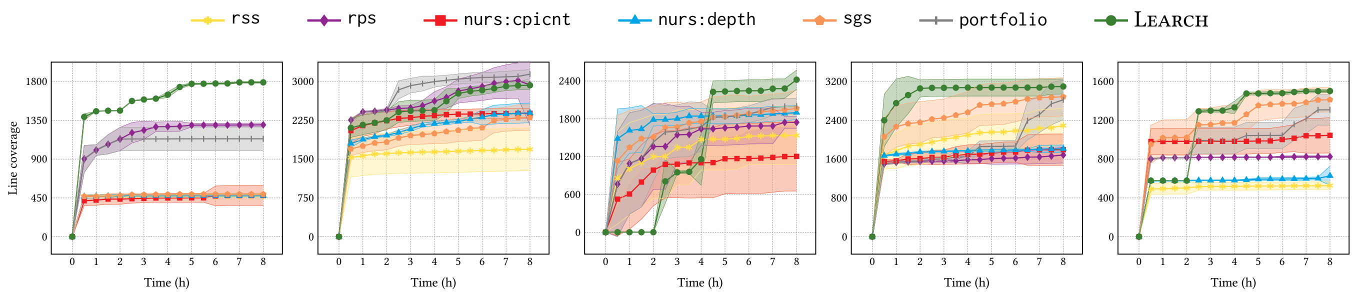
Different from Algorithm 1 which selects a state per branch, the original KLEE performs state selection per instruction. In our initial experiments on `coreutils`, we found that running the heuristics with Algorithm 1 gave better results and thus used Algorithm 1 for all heuristics in our evaluation. 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].

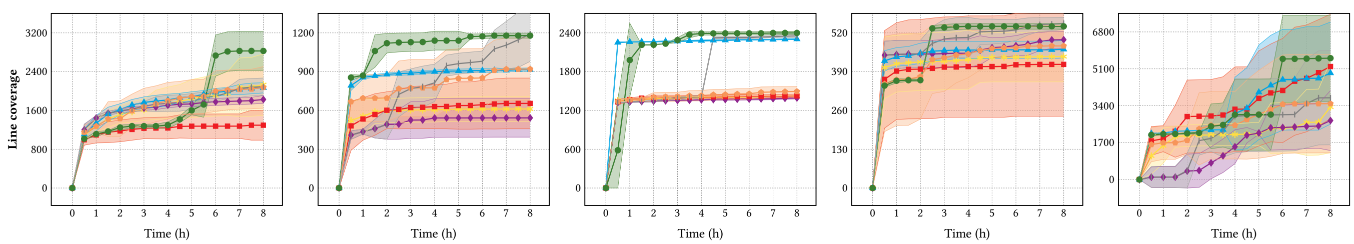



**(a) Line coverage of all files in the package.** **(b) Number of programs with the best coverage.**

**Figure 5: Line coverage of running KLEE with different strategies for 1h on the 52 `coreutils` testing programs. The numbers were averaged over 20 runs and the error bars represent standard deviations.**




**(a) Line coverage for `diff`.** **(b) Line coverage for `find`.** **(c) Line coverage for `grep`.** **(d) Line coverage for `gawk`.** **(e) Line coverage for `patch`.**




**(f) Line coverage for `objcopy`.** **(g) Line coverage for `readelf`.** **(i) Line coverage for `make`.** **(j) Line coverage for `cjson`.** **(k) Line coverage for `sqlite`.**

**Figure 6: Line coverage for the 10 real-world programs by running KLEE with different strategies for 8h. Mean values and standard deviations over 20 runs are plotted.**

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

**Table 3: Percentage of covered MainLOCs.**

| | rss | rps | nurs:cpicnt | nurs:depth | sgs | portfolio | Learch |
|---|---|---|---|---|---|---|---|
| coreutils | 66.4 | 73.1 | 71.6 | 71.4 | 72.0 | 75.4 | 76.9 |
| diff | 30.3 | 53.7 | 31.1 | 30.6 | 32.3 | 50.5 | 59.1 |
| find | 52.1 | 57.7 | 58.3 | 56.0 | 60.2 | 61.2 | 61.0 |
| grep | 21.8 | 29.7 | 17.1 | 28.1 | 29.8 | 29.7 | 36.5 |
| gawk | 39.2 | 39.2 | 39.2 | 43.0 | 39.2 | 43.0 | 39.2 |
| patch | 13.8 | 19.1 | 24.4 | 15.1 | 33.5 | 31.9 | 35.8 |
| objcopy | 9.9 | 9.5 | 6.0 | 9.3 | 8.3 | 9.8 | 13.3 |
| readelf | 4.8 | 3.9 | 5.2 | 6.5 | 7.7 | 9.1 | 9.0 |
| make | 33.3 | 33.3 | 33.0 | 45.2 | 33.3 | 45.2 | 45.2 |
| cjson | 79.8 | 80.2 | 76.5 | 79.6 | 79.7 | 80.3 | 80.3 |
| sqlite | 8.1 | 6.3 | 12.8 | 11.4 | 9.2 | 8.7 | 14.2 |

**Table 4: Percentage of covered ELOCs.**

| | rss | rps | nurs:cpicnt | nurs:depth | sgs | portfolio | Learch |
|---|---|---|---|---|---|---|---|
| coreutils | 13.6 | 15.2 | 14.7 | 14.9 | 14.9 | 15.7 | 16.1 |
| diff | 1.9 | 3.3 | 2.0 | 2.0 | 2.0 | 3.1 | 3.6 |
| find | 1.0 | 1.1 | 1.2 | 1.1 | 1.2 | 1.2 | 1.2 |
| grep | 2.3 | 2.5 | 1.8 | 2.9 | 3.1 | 3.1 | 3.9 |
| gawk | 0.9 | 0.9 | 0.9 | 1.0 | 0.9 | 1.0 | 0.9 |
| patch | 1.7 | 2.4 | 3.1 | 1.9 | 4.2 | 4.0 | 4.5 |
| objcopy | 0.5 | 0.4 | 0.3 | 0.4 | 0.4 | 0.5 | 0.6 |
| readelf | 1.5 | 1.2 | 1.6 | 2.0 | 2.5 | 2.9 | 2.9 |
| make | 3.2 | 3.2 | 3.2 | 4.4 | 3.2 | 4.4 | 4.4 |
| cjson | 7.3 | 7.4 | 7.0 | 7.3 | 7.3 | 7.4 | 7.4 |
| sqlite | 5.5 | 4.2 | 8.8 | 7.8 | 6.3 | 5.9 | 9.7 |

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 heuristics 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.

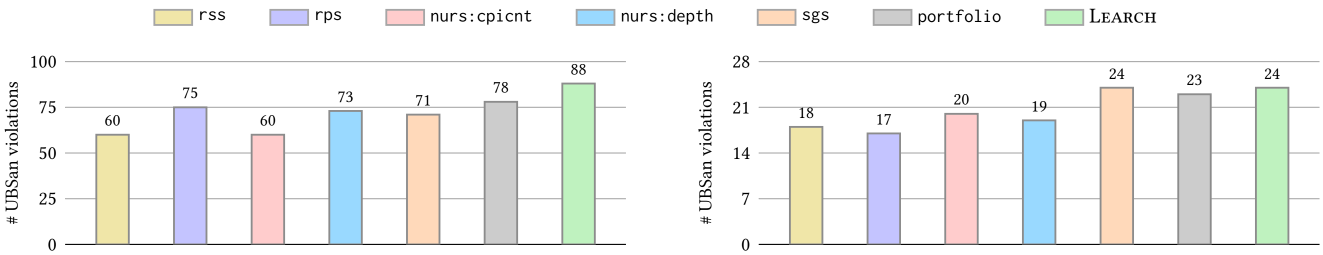
**Percentage of covered lines.** Apart from absolute line coverage, we calculated the percentage of covered lines to investigate how thoroughly the test programs are covered by KLEE and the strategies. We measured the percentage of covered MainLOCs and ELOCs (mean value from 20 runs). The total number 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].

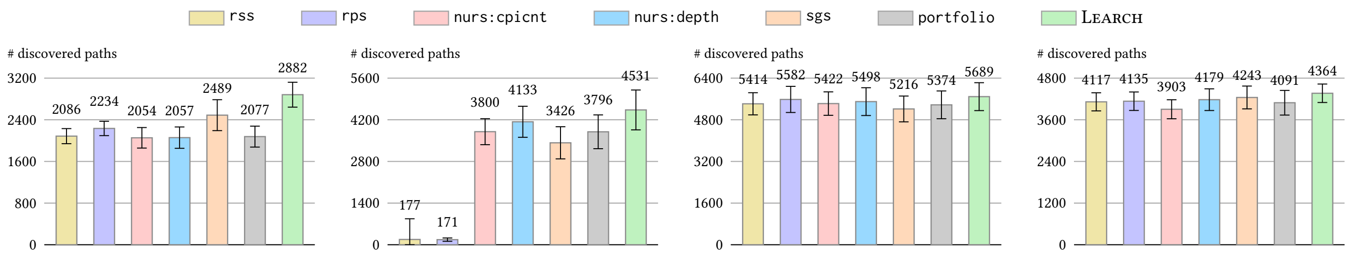
**Summary on generalizability.** Learch's effectiveness in code coverage demonstrates its generalizability: it generalizes between programs from the same package (i.e., training and testing on `coreutils`). This is because these programs usually share the same code and our learning method can capture 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.




(a) Results for the 52 `coreutils` testing programs.

(d) Results for the 10 real-world programs.

**Figure 7: The total number of detected UBSan violations found by KLEE with different strategies.**

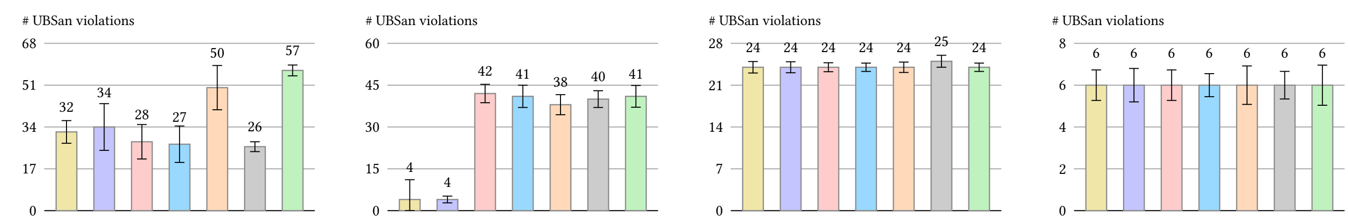



(a) # discovered paths for `objcopy`.

(b) # discovered paths for `readelf`.

(c) # discovered paths for `make`.

(d) # discovered paths for `sqlite`.




(e) # UBSan violations for `objcopy`.

(f) # UBSan violations for `readelf`.

(g) # UBSan violations for `make`.

(h) # UBSan violations for `sqlite`.

**Figure 8: The number of paths and UBSan violations discovered by AFL after 8h using KLEE tests as initial seeds. The numbers were averaged over 20 runs and the error bars represent standard deviations.**

## 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, 31, 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

```c
static bool consider_arm_swap (struct predicate *p) {
  ...
  pr = &p->pred_right;
  // findutils-4.7.0/find/tree.c: line 538
  pl = &p->pred_left->pred_right;
  ...
```

**Figure 9: A null pointer dereference.**

```c
1 const char * _bfd_coff_read_string_table (bfd *abfd) {
2   ...
3   // binutils-2.36/bfd/coffgen.c: line 1676
4   pos += obj_raw_syment_count (abfd)
5            * bfd_coff_symesz (abfd);
6   ...
```

**Figure 10: Overflows leading to an incorrect file position.**

```c
1 static char * find_map_unquote (...) {
2   ...
3   // make-4.3/src/read.c: line 2354
4   memmove (&p[i], &p[i/2],
5     (string_len - (p - string)) - (i/2) + 1);
6   ...
7 }
```

**Figure 11: An overflow leading to wrong array accesses.**

```c
# define ISDIGIT(c) ((unsigned int) (c) - '0' <= 9)
...
// coreutils-8.31/lib/strnumcmp-in.h: line 224
for (log_a = 0; ISDIGIT (tmpa); ++log_a)
  do { tmpa = *++a; }
  while (tmpa == thousands_sep);
...
```

**Figure 12: A benign overflow.**

[30, 31, 42]. In Figure 8(a) to Figure 8(d), we show the number of paths discovered by AFL for the four programs, respectively. When using the initial seeds from Learch, AFL discovered the most number of paths for all four programs. With the initial seeds from `rss` and `rps`, AFL discovered very few paths for `readelf`, which is not surprising as `rss` and `rps` achieved very low coverage on `readelf` with KLEE.

**Detecting security violations.** We re-ran the tests generated by AFL with UBSan checkers turned on. The number of detected violations is shown in Figure 8(e) to Figure 8(h). Overall, with initial seeds from Learch, AFL found 128 violations in total, outperforming other heuristics (10 more than the best heuristic, `sgs`). We provide a manual analysis of the detected violations next.

## 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.

**Table 5: Our bug reports to the developers. The programs without any manually identified bugs are not listed.**

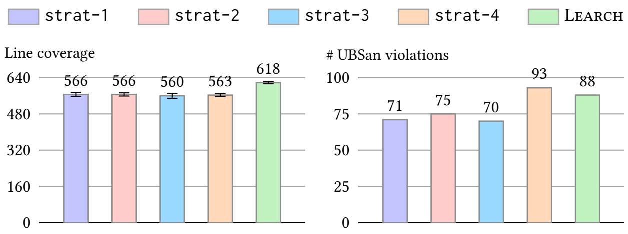
| | Violations | Bug reports | Confirmed | Fixed/Will fix |
|---|---|---|---|---|
| `coreutils` | 88 | 3 | 2 | 2 |
| `find` | 5 | 2 | 2 | 2 |
| `objcopy` | 73 | 29 | 3 | 3 |
| `readelf` | 57 | 5 | 1 | 1 |
| `make` | 26 | 3 | 3 | 3 |
| `sqlite` | 9 | 4 | 2 | 0 |
| Total | 258 | 46 | 13 | 11 |

**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 deference 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

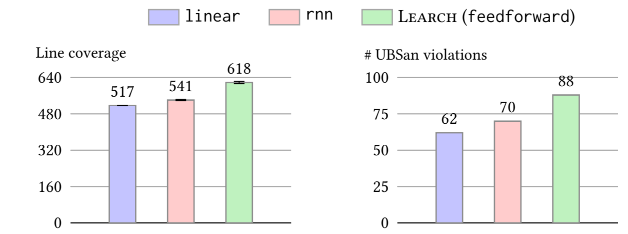
We investigate the usefulness of Learch's design choices. Due to space limit, we mainly present the results on `coreutils`. For the real-world programs, we observed the same phenomenon as `coreutils`.

**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




**(a) Line coverage for the whole package.** **(b) Number of UBSan violations.**

**Figure 13: Results of learned strategies on `coreutils` test set.**




**(a) Line coverage for the whole package.** **(b) Number of UBSan violations.**

**Figure 14: Results of different models on `coreutils` test set.**

Figure 13. The individual strategies already achieved more coverage (~20 lines) than the individual manual heuristics. Even though the absolute coverage numbers were similar, the four strategies covered different parts of the program. As a result, LEARCH, combined from the four strategies, was the most performant overall. This is the desired outcome of the iterative learning in Algorithm 4.

LEARCH also found more UBSan violations than `strat-1`, `strat-2`, and `strat-3`, as a result of the combination. LEARCH found 5 fewer violations than `strat-4` because `strat-4` found many violations after 15m. Therefore, to make LEARCH detect more violations, we can simply increase the time budget.

**Different choices of machine learning model.** Other than feedforward networks used in LEARCH, we considered simpler linear regression (`linear`) and more complicated recurrent neural networks (`rnn`). For `rnn`, we added a hidden state of dimension 64 between a state and its parent. We trained `linear` and `rnn` on the same supervised dataset as LEARCH, and ran them with the same configuration (i.e., four independent runs each taking a quarter of the time budget) as LEARCH on our test set. The results on the `coreutils` test set are shown in Figure 14, showing that LEARCH outperformed `linear` and `rnn`. The reason is likely that the complexity of feedforward networks is well-suited for our learning task.

## 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].

**Concolic testing and fuzzing.** Concolic testing and fuzzing are different approaches for program testing but can benefit from advances in symbolic execution because many of them use symbolic execution for triggering complex paths. Concolic testing [33, 57, 58, 62] concretely executes the program alongside symbolic execution and negates the path constraint of visited branches to produce new tests covering unvisited branches. Heuristics have been learned for selecting branches in concolic testing [19, 20]. Fuzzing is a technique that concretely executes the program and generates concrete inputs based on input specifications [11, 40, 47] or mutations from existing inputs [1, 7, 12, 26, 30, 31, 41, 42, 45, 65, 72]. Symbolic execution has been used for improving fuzzing [29, 54]. Hybrid testing [27, 69, 75] combines concolic testing and fuzzing in an alternative manner to benefit from the advantages of both.

**Machine learning for program analysis and security.** Machine learning has been extensively used for security tasks. Markov chain [12], feedforward networks [65], recurrent networks [35], imitation learning [37], reinforcement learning [72] have been used for improving test generation in fuzzing. The authors of [73] leverage reinforcement learning for directed symbolic execution. Many other tasks such as binary analysis [38], malware analysis [60], and taint analysis [64] have been solved by data-driven approaches.

## 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.

## REFERENCES


[1] 2021. American fuzzy lop. https://lcamtuf.coredump.cx/afl/

[2] 2021. Gcov (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[3] 2021. Google Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite

[4] 2021. UndefinedBehaviorSanitizer — Clang 6 documentation. https://releases.llvm.org/6.0.0/tools/clang/docs/UndefinedBehaviorSanitizer.html

[5] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *EuroS&P 2017*. https://doi.org/10.1109/EuroSP.2017.11

[6] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. 2008. Finding Bugs in Dynamic Web Applications. In *ISSTA 2008*. https://doi.org/10.1145/1390630.1390662

[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *NDSS 2019*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[8] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *ICSE 2014*. https://doi.org/10.1145/2568225.2568293

[9] Mislav Balunovic, Pavol Bielik, and Martin Vechev. 2018. Learning to Solve SMT Formulas. In *NeurIPS 2018*. https://proceedings.neurips.cc/paper/2018/hash/68331ff0427b551b68e911eebe35233b-Abstract.html

[10] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *USENIX Security 2014*. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao

[11] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security 2019*. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *CCS 2017*. https://doi.org/10.1145/3133956.3134020

[13] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS 2008*. https://doi.org/10.1007/978-3-540-78800-3_27

[14] Suhabe Bugrara and Dawson R. Engler. 2013. Redundant State Detection for Dynamic Symbolic Execution. In *USENIX ATC 2013*. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bugrara

[15] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *ISSTA 2020*. https://doi.org/10.1145/3395363.3397360

[16] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI 2008*. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[17] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: automatically generating inputs of death. In *CCS 2006*. https://doi.org/10.1145/1180405.1180445

[18] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Communications of ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[19] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *ICSE 2018*. https://doi.org/10.1145/3180155.3180166

[20] Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics. In *FSE 2019*. https://doi.org/10.1145/3338906.3338964

[21] Sooyoung Cha and Hakjoo Oh. 2020. Making Symbolic Execution Promising by Learning Aggressive State-pruning Strategy. In *ESEC/FSE 2020*. https://doi.org/10.1145/3368089.3409755

[22] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *S&P 2012*. https://doi.org/10.1109/SP.2012.31

[23] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *S&P 2017*. https://doi.org/10.1109/SP.2017.40

[24] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification. In *NDSS 2019*. https://www.ndss-symposium.org/ndss-paper/analyzing-semantic-correctness-with-symbolic-execution-a-case-study-on-pkcs1-v1-5-signature-verification/

[25] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to Accelerate Symbolic Execution via Code Transformation. In *ECOOP 2018*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.6

[26] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *S&P 2018*. https://doi.org/10.1109/SP.2018.00046

[27] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *S&P 2020*. https://doi.org/10.1109/SP40000.2020.00002

[28] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008*. https://doi.org/10.1007/978-3-540-78800-3_24

[29] Julian Fietkau, Bhargava Shastry, and Jean-Pierre Seifert. 2017. KleeFL - Seeding Fuzzers With Symbolic Execution. In *Posters presented at USENIX Security 2017*. https://github.com/julieeen/kleefl/raw/master/USENIX2017poster.pdf

[30] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security 2020*. https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[31] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *S&P 2018*. https://doi.org/10.1109/SP.2018.00040

[32] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *CAV 2007*. https://doi.org/10.1007/978-3-540-73368-3_52

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI 2005*. https://doi.org/10.1145/1065010.1065036

[34] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS 2008*. https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[35] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *ASE 2017*. https://doi.org/10.1109/ASE.2017.8115618

[36] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial Symbolic Execution for Detecting Concurrency-related Cache Timing Leaks. In *FSE 2018,*. https://doi.org/10.1145/3236024.3236028

[37] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *CCS 2019*. https://doi.org/10.1145/3319535.3363230

[38] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *CCS 2018*. https://doi.org/10.1145/3243734.3243866

[39] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed selection for successful fuzzing. In *ISSTA 2021*. https://doi.org/10.1145/3460319.3464795

[40] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security 2012*. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[41] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *S&P 2020*. https://doi.org/10.1109/SP40000.2020.00063

[42] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-Fuzzing Techniques. In *USENIX Security 2019*. https://www.usenix.org/conference/usenixsecurity19/presentation/jung

[43] Timotej Kapus, Frank Busse, and Cristian Cadar. 2020. Pending Constraints in Symbolic Execution for Better Exploration and Seeding. In *ASE 2020*. https://ieeexplore.ieee.org/document/9286054

[44] James C. King. 1976. Symbolic Execution and Program Testing. *Communications of ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[45] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *CCS 2018*. https://doi.org/10.1145/3243734.3243804

[46] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *PLDI 2012*. https://doi.org/10.1145/2254064.2254088

[47] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI 2014*. https://doi.org/10.1145/2594291.2594334

[48] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *OOPSLA 2013*. https://doi.org/10.1145/2509136.2509553

[49] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS 2016*. https://doi.org/10.1145/2976749.2978309

[50] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *SAS 2011*. https://doi.org/10.1007/978-3-642-23702-7_11

[51] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security 2020*. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[52] Corina S. Pasareanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE 2010*. https://doi.org/10.1145/1858996.1859035

[53] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. 2015. Analyzing Protocol Implementations for Interoperability. In *NSDI 2015*. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pedrosa

[54] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *S&P 2018*. https://doi.org/10.1109/SP.2018.00056

[55] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *S&P 2020*. https://doi.org/10.1109/SP40000.2020.00024

[56] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed Incremental Symbolic Execution. In *PLDI 2011*. https://doi.org/10.1145/1993498.1993558

[57] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic Execution with SymCC: Don't Interpret, Compile!. In *USENIX Security 2020*. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[58] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based Symbolic Execution for Binaries. In *NDSS 2021*. https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/

[59] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Security 2015*. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[60] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. 2001. Data Mining Methods for Detection of New Malicious Executables. In *S&P 2001*. https://doi.org/10.1109/SECPRI.2001.924286

[61] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2019. Learning a SAT Solver from Single-Bit Supervision. In *ICLR 2019*. https://openreview.net/forum?id=HJMC_iA5tm

[62] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a Concolic Unit Testing Engine for C. In *FSE 2005*. https://doi.org/10.1145/1081706.1081750

[63] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[64] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. 2020. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. In *S&P 2020*. https://doi.org/10.1109/SP40000.2020.00022

[65] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *S&P 2019*. https://doi.org/10.1109/SP.2019.00052

[66] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. In *NDSS 2019*. https://www.ndss-symposium.org/ndss-paper/neuro-symbolic-execution-augmenting-symbolic-execution-with-neural-constraints/

[67] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *USENIX Security 2015*. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

[68] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *S&P 2016*. https://doi.org/10.1109/SP.2016.17

[69] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS 2016*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[70] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *ICSE 2018*. https://doi.org/10.1145/3180155.3180251

[71] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *S&P 2017*. https://doi.org/10.1109/SP.2017.23

[72] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed for Greybox Fuzzing. In *NDSS 2021*. https://www.ndss-symposium.org/ndss-paper/reinforcement-learning-based-hierarchical-seed-scheduling-for-greybox-fuzzing/

[73] Jie Wu, Chengyu Zhang, and Geguang Pu. 2020. Reinforcement Learning Guided Symbolic Execution. In *SANER 2020*. https://doi.org/10.1109/SANER48275.2020.9054815

[74] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *DSN 2009*. https://doi.org/10.1109/DSN.2009.5270315

[75] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security 2018*. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[76] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. 2018. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. In *MICRO 2018*. https://doi.org/10.1109/MICRO.2018.00071

[77] Yufeng Zhang, Zhenbang Chen, Ziqi Shuai, Tianqi Zhang, Kenli Li, and Ji Wang. 2020. Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once. In *ASE 2020*. https://doi.org/10.1145/3324884.3416645