SymTuner: Maximizing the Power of Symbolic Execution by Adaptively Tuning External Parameters

Sooyoung Cha
Sungkyunkwan University
Republic of Korea
sooyoung.cha@skku.edu

Myungho Lee
Korea University
Republic of Korea
myungho_lee@korea.ac.kr

Seokhyun Lee
Korea University
Republic of Korea
seokhyunlee@korea.ac.kr

Hakjoo Oh*
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

ABSTRACT

We present SymTuner, a novel technique to automatically tune external parameters of symbolic execution. Practical symbolic execution tools have important external parameters (e.g., symbolic arguments, seed input) that critically affect their performance. Due to the huge parameter space, however, manually customizing those parameters is notoriously difficult even for experts. As a consequence, symbolic execution tools have typically been used in a suboptimal manner that, for example, simply relies on the default parameter settings of the tools and loses the opportunity for better performance. In this paper, we aim to change this situation by automatically configuring symbolic execution parameters. With SymTuner that takes parameter spaces to be tuned, symbolic executors are run without manual parameter configurations; instead, appropriate parameter values are learned and adjusted during symbolic execution. To achieve this, we present a learning algorithm that observes the behavior of symbolic execution and accordingly updates the sampling probability of each parameter space. We evaluated SymTuner with KLEE on 12 open-source C programs. The results show that SymTuner increases branch coverage of KLEE by 56% on average and finds 8 more bugs than KLEE with its default parameters over the latest releases of the programs.

ACM Reference Format:
Sooyoung Cha, Myungho Lee, Seokhyun Lee, and Hakjoo Oh. 2022. SymTuner: Maximizing the Power of Symbolic Execution by Adaptively Tuning External Parameters. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510185

1 INTRODUCTION

Decades of research have transformed symbolic execution into a mainstream technique in software testing. The basic idea of symbolic execution is to replace program inputs by symbolic variables and explore the execution paths of a program symbolically. Since its inception [7, 32, 41], symbolic execution has been an active research area [5]. In particular, the last decade has seen remarkable advances, significantly mitigating main challenges such as path explosion and constraint solving [6, 14, 23, 34, 36, 38, 39, 45, 48, 52, 54, 66]. Equipped with these techniques, symbolic execution tools such as KLEE [11] have become publicly available and widely used in both academia and industry [12, 13].

```
$ klee --simplify-sym-indices --max-memory=1000 --optimize
      --use-forked-solver --use-cex-cache --external-calls=all
      --max-sym-array-size=4096 --max-instruction-time=30s
      --max-time=60min --max-memory-inhibit=false
      --max-static-fork-pct=1 --max-static-solve-pct=1
      --max-static-cpfork-pct=1 --switch-type=internal
      --search=random-path --search=nurs:covnew
      --batch-instructions=10000 ./pgm.bc --sym-args 0 1 10
      --sym-args 0 2 2 --sym-files 1 8 --sym-stdin 8 --sym-stdout ...
```

Figure 1: The parameter setting of KLEE used in [49]

Despite the progress, however, maximizing the performance of modern symbolic execution tools is notoriously difficult in practice. One main reason is that state-of-the-art symbolic executors have a number of important external parameters that critically affect their effectiveness. For example, Figure 1 shows a typical command for running KLEE, which was tailored to GNU Coreutils by the original authors of KLEE [11, 49]. These parameters, for example, are to select which search strategy to use, to decide the symbolic arguments, and to choose the memory budget. It is well-known that these parameters have a huge impact on the runtime performance of symbolic execution and therefore must be carefully tuned for each target program [45, 60, 63].

Manually tuning such parameters is challenging even for experts. Note that more than half of the parameters in Figure 1 are of non-boolean types (e.g., string or integer), and their combination induces an enormous search space. Consequently, KLEE has typically been used without proper configuration of those parameters; existing works either (1) simply rely on the parameter setting in Figure 1 [17, 18, 23, 50, 63] (even for programs beyond Coreutils) or (2) manually tune specific parameters (e.g., symbolic arguments) atop the default parameter values [37, 45, 52, 60, 66]. Recently, a few techniques [14, 16] have been proposed for tuning search heuristics automatically but other critical parameters still need to be configured manually.

In this paper, we present SymTuner, a novel technique for automatically tuning symbolic execution parameters. Initially, from the users, SymTuner takes as input the sample spaces for the parameters to be tuned. Then, with SymTuner, symbolic execution tools such as KLEE can be run without manual parameter tuning; appropriate parameter values for the target program are automatically adjusted by SymTuner during symbolic execution. To do so, along

*Corresponding author

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510185

the symbolic execution process, SymTuner uses a custom learning algorithm that repeatedly samples a set of parameter values from the sample spaces, evaluates the performance of symbolic execution with the sampled values, and refines the probability distributions of the sample spaces based on the evaluation result.

Experimental results show that SymTuner remarkably enhances symbolic execution in terms of both code coverage and bug-finding. We applied SymTuner to KLEE [11], a representative symbolic execution tool for C programs, and evaluated it on the latest versions of 12 GNU open-source programs (ranging from 5K to 161K LoC). KLEE with SymTuner covered 56% and 31% more branches on average than conventional KLEE with its default parameter values and the parameter setting in Figure 1, respectively. Also, SymTuner enabled KLEE to discover 11 different bugs that cause the latest versions of the open-source programs to crash, far outweighing the conventional KLEE that ended up finding three of them. Compared to KLEE with a naive approach that randomly samples parameter values, SymTuner succeeded in increasing the number of covered branches and found bugs by 12% and 45%, respectively. We also show that our approach is also applicable to CREST [21], a tool for concolic testing [13, 25, 53], another major approach of symbolic execution.

Contributions. We summarize our contributions below:

- We present SymTuner, a new technique for automatically tuning diverse parameters of symbolic execution. The key technical contribution is the domain-specific learning algorithm for symbolic execution, which observes the behavior of symbolic execution with randomly sampled parameters and gradually learns to sample effective parameter values.
- We conduct extensive evaluation of SymTuner on 12 GNU open-source programs. We make our tool, SymTuner, open-sourced and the benchmarks publicly available.[1]

2 PRELIMINARIES

2.1 Basic Symbolic Execution

Symbolic execution explores the execution paths of a program by maintaining a set of program states, where a state is a triplet $(instr, store, \Phi)$ of an instruction ($instr$) to be executed, a symbolic memory store ($store$) mapping program variables to symbolic values, and a path condition (Φ) that is a sequence of symbolic branches representing the path exercised by the current state.

Algorithm 1 describes the overall algorithm. It takes as input a program (pgm), a testing budget ($budget$), and a vector of parameter values (V). For the moment, let us ignore the last input; the role of the parameters (V) will be described in Section 2.2.

At line 2, the algorithm creates the set of initial states, i.e., a singleton set of the initial state $s_0 = (instr_0, store_0, true)$, where $instr_0$ denotes the first instruction of the program and $store_0$ is the initial symbolic memory. The set T of test cases is initially empty (line 3). After initializing S and T, symbolic execution goes into the loop at lines 4–15.

At line 5, symbolic execution selects a state s from S to navigate deeper into the program, and removes s from S (line 6). At line 7, the instruction in the current state $(instr, store, \Phi)$ is executed,

[1] SymTuner: https://github.com/skkusal/symtuner

Algorithm 1 Symbolic execution

Input: Program (pgm), budget ($budget$), and parameter values (V).
Output: A set of test cases (T)

```
1:  procedure SymExecutor(pgm, budget, V)
2:      S ← {s_0}                                   ▷ s_0 = (instr_0, store_0, true)
3:      T ← ∅                                       ▷ test cases
4:      repeat
5:          s ← Choose(S)                           ▷ s = (instr, store, Φ)
6:          S ← S \ {s}
7:          s' ← Execute(s)                         ▷ s' = (instr', store', Φ)
8:          if instr' is a branch whose condition is φ then
9:              if SAT(Φ ∧ φ) then                  ▷ true branch is reachable
10:                 S ← S ∪ {(instr^1, store', Φ ∧ φ)}
11:             if SAT(Φ ∧ ¬φ) then                 ▷ false branch is reachable
12:                 S ← S ∪ {(instr^2, store', Φ ∧ ¬φ)}
13:         else if instr' is a halt instruction then
14:             T ← T ∪ {Model(Φ)}                  ▷ generate a test case
15:     until budget expires (or S = ∅)
16:     return T
```

producing the next state $(instr', store', \Phi)$. If $instr'$ is a branch instruction whose condition is ϕ, the algorithm checks whether the both sides of the branch are reachable from the current state. If the true branch is reachable (i.e., $\text{SAT}(\Phi \wedge \phi)$), we add the updated state $(instr^1, store', \Phi \wedge \phi)$ to S (line 10), where $instr^1$ denotes the first instruction in the true branch. Similarly, we add the updated state for the false branch to S when the path condition, i.e., $(\Phi \wedge \neg\phi)$, is satisfiable (line 12). When $instr'$ is a halt instruction (line 13), a test case is generated from the model of the current path condition. The SymExecutor procedure repeats the process described so far until the given budget expires. Upon termination, the set T of test cases is returned.

2.2 Parameters of Symbolic Execution

Although the basic algorithm is simple, real-world symbolic execution tools involve various parameters that have a critical impact on the performance of Algorithm 1. For example, symbolic executors such as KLEE [11] take a parameter that determines which program inputs to be replaced by symbolic variables (e.g., `--sym-args` in Figure 1); in Algorithm 1, the initial symbolic memory ($store_0$) is defined by this parameter value. Another example is a parameter that specifies the maximum memory capacity available (e.g., `--max-memory` in Figure 1), which is an important factor in practice as state explosion frequently occurs when running symbolic execution on sizable programs. The Choose function itself at line 5 is also a parameter, called search heuristic [14, 45, 54], and users of KLEE can choose from 10 different options (e.g., `--search=nurs:covnew` in Figure 1) and interleave them. The constraint solver used by the SAT and Model functions in Algorithm 1 can be configured as well; for example, users can decide which SMT solver to use and fine-tune their behavior. Figure 1 shows that KLEE also provides various parameters, including `max-instruction-time` and `batching-instructions`, which have a large space (e.g., integer).

Manually tuning those parameter values is so nontrivial that it has been typical to use symbolic execution without proper configuration [17, 18, 23, 45, 50, 52, 60, 63, 66]. The goal of this paper is to

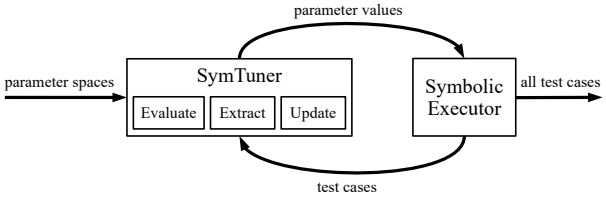

Figure 2: Overview of symbolic execution with SymTuner

change this unfortunate practice by automating the task of tuning parameters of symbolic execution.

3 OUR APPROACH

Figure 2 illustrates how SymTuner and symbolic execution interact. Initially, SymTuner takes k predefined parameter spaces as input, where a parameter space prescribes possible values that the parameter can take on. At a high-level, SymTuner iteratively samples k parameter values based on the learned probability distributions of the parameter spaces, and runs symbolic execution with the sampled parameter values. As output of symbolic execution, a set of test cases is generated, which is in turn used for adjusting the probability distributions of the sampling spaces. During the first few times, SymTuner focuses on exploration by running symbolic execution with various random parameter values. Once an adequate number of test cases is collected, SymTuner starts to update the sampling probabilities and exploit the learned knowledge.

SymTuner works in three phases: Evaluate, Extract, and Update. The first step produces learning data by evaluating the quality of the parameter values used for generating each test case in terms of both code coverage and found bugs. The aim of the second step (Extract) is to identify meaningful data from the total amount of data generated by the first step. Using the extracted data, the last step (Update) updates the probability of how SymTuner samples from the k parameter spaces. By repeating the above process which interleaves SymTuner and symbolic executor, the sampling probabilities are gradually updated. Upon termination (i.e., hitting a time limit), the set of all test cases generated so far are returned.

3.1 Parameter Space

Basically, we define the sample space of each parameter to be discrete rather than continuous to circumvent unnecessarily large search space. For instance, suppose that we define the parameter space for memory budget (MB) as all integers between 100 and 2000. On the basis of a parameter value, e.g., 1000MB, in the space, the adjacent values such as 999MB and 1001MB are unlikely to have a significant impact on the performance of symbolic execution compared to the farther values (e.g., 500MB, 2000MB); thus, our sample space for each parameter is discrete to maintain only the values which are likely to affect performance.

In our approach, we assume that k predefined parameter spaces, denoted $\mathbb{S} = S_1 \times S_2 \times \cdots \times S_k$, are given. Each space S_i $(1 \le i \le k)$ for the i-th parameter consists of two components:

$$S_i = (SV_i, \eta_i^{\max})$$

where $SV_i = \{sv_i^1, sv_i^2, \cdots, sv_i^n\}$ is the set of possible parameter values for S_i and $\eta_i^{\max}$ denotes the maximum number of times to sample from the set SV_i. For most parameter spaces, $\eta^{\max}$ is 1 as a parameter value is typically used only once during a single run of symbolic execution (e.g., `--max-memory` in Figure 1). In general, however, it can be bigger than 1. For example, the value of $\eta^{\max}$ for symbolic arguments can be any natural number as we can use multiple symbolic arguments with different size in a single run (e.g., `--sym-args` in Figure 1). For instance, the parameter space for symbolic arguments can be given as follows:

$$S_{args} = (\{2, 4, 6, 8\}, 3).$$

which means we can select up to three symbolic arguments, where each argument has the length of one of the elements in $\{2, 4, 6, 8\}$.

3.1.1 ***Sample.*** To sample from each predefined space S_i, let us define and use the following Sample function:

$$\mathsf{Sample}(S_i, \mathcal{P}_i^c, \mathcal{P}_i) = PV_i$$

The inputs are a parameter space S_i and two probability functions, $\mathcal{P}_i$ and $\mathcal{P}_i^c$. The former $\mathcal{P}_i : SV_i \to [0, 1]$ denotes the sampling probability of each parameter value in SV_i and the latter $\mathcal{P}_i^c : [1, \eta_i^{\max}] \to [0, 1]$ represents the probability for the sampling times.

We denote the output of Sample by $PV_i \in \wp(SV_i)$, which is a set of sampled parameter values. Here, we allow PV_i to be a multiset that has duplicated elements and we abuse the notation $\wp(SV_i)$ to denote the set of all sub-multisets of SV_i. To obtain PV_i from the space S_i, the Sample function first determines the number m of sampling times based on the probability $\mathcal{P}_i^c$. Then, we sample a candidate value from SV_i for m times using the probability $\mathcal{P}_i$ and add the value to PV_i. For instance, when the Sample function takes the parameter space S_{args} shown above, the possible outcomes of Sample, i.e., PV_{args}, are as follows:

$$\{\!\{2\}\!\}, \{\!\{6\}\!\}, \{\!\{8, 6\}\!\}, \{\!\{4, 8\}\!\}, \{\!\{2, 4, 2\}\!\}, \{\!\{4, 4, 4\}\!\}, \cdots$$

where $\{\!\{\}\!\}$ denotes a multiset that allows duplicated elements.

Since the two probabilities in the Sample function determine the parameter values used when performing symbolic execution on the target program, the most important question is how to learn these probabilities, $\mathcal{P}_i^c$ and $\mathcal{P}_i$, for each parameter space S_i $(1 \le i \le k)$ to maximize the performance of symbolic execution. To resolve this, SymTuner learns those probabilities based on the data accumulated during symbolic execution.

3.2 Symbolic Execution with SymTuner

To learn parameter values, we perform symbolic execution multiple times with diverse parameter values by dividing the total time budget into smaller budgets. It enables SymTuner to interact with symbolic executor multiple times during the given time budget, and to gradually find more effective parameter values through the multiple interactions. We describe how SymTuner interacts with symbolic executor (Algorithm 2) in detail.

Algorithm 2 takes as input a program, a budget, and k predefined parameter spaces. At lines 1, the algorithm first initializes each of the following four components to an empty set or vector: the vector $V \in \wp(SV_1) \times \wp(SV_2) \times \cdots \times \wp(SV_k)$ of parameter values (recall $\wp(SV_i)$ denotes the set of all sub-multisets of SV_i), the set D of learning data, the set T of test cases, and the set *TotalT* of

Algorithm 2 Symbolic execution with SymTuner

Input: Program (*pgm*), budget (*budget*), k parameter spaces ($\mathbb{S}$).
Output: Test cases (*TotalT*)

1: $\langle V, D, T, TotalT \rangle \leftarrow \langle \langle\rangle, \emptyset, \emptyset, \emptyset \rangle$ ▷ D is learning data
2: $flag \leftarrow 0$
3: $budget_s \leftarrow budget * \eta_{\text{ratio}}$ ▷ $\eta_{\text{ratio}} = 0.5\%$
4: **repeat**
5: **for** $i = 1$ **to** η_{step} **do** ▷ $\eta_{\text{step}} = 20$
6: $V, D \leftarrow \text{SymTuner}(T, \mathbb{S}, V, D, flag)$
7: $T \leftarrow \text{SymExecutor}(pgm, budget_s, V)$
8: $TotalT \leftarrow TotalT \cup T$
9: $flag \leftarrow 1$
10: $budget_s \leftarrow budget_s * 2$
11: **until** *budget* expires
12: **return** *TotalT*

accumulated test cases. Then, the *flag* value is also initially set to 0 (line 2); the value of 0 makes SymTuner focus only on exploration by trying diverse parameter values in the k parameter spaces. At the next line, the algorithm initializes the time budget $budget_s$ for a single run of symbolic execution by multiplying the total testing budget *budget* and the hyper-parameter η_{ratio}. In the experiments, we set the hyper-parameter η_{ratio} to a small value such as 0.005.

At lines 5–10, symbolic executor and SymTuner iteratively interact with each other by exchanging test cases and parameter values. At line 6, SymTuner takes five input values—test cases (T), parameter spaces ($\mathbb{S}$), previously used parameter values (V), learning data (D), and flag (*flag*) — and returns accumulated learning data with newly sampled parameter values; at first, SymTuner generates the parameter values by randomly sampling from the predefined spaces $\mathbb{S}$ (line 6). Then, the SymExecutor is run with the program, the current time budget, and the new parameter values. As output of symbolic execution, a set T of test cases is produced (line 7). At line 8, we accumulate T in the set *TotalT* of total test cases. After the first interaction between SymTuner and SymExecutor is repeated η_{step} times (lines 5–8), the algorithm sets *flag* to 1, which indicates that SymTuner is ready to perform online learning. In experiments, we set $\eta_{\text{step}} = 20$. At line 10, it doubles the size of the time budget $budget_s$ as more learning data accumulation increases the confidence in how to tune parameter values. The outer loop repeats until the total time budget (*budget*) is exhausted. Upon termination, the algorithm returns as the final output the accumulated test cases.

3.3 SymTuner

Algorithm 3 describes how SymTuner generates the set D of learning data from the set T of test cases and how it updates the sampling probabilities of the parameter spaces based on D. SymTuner works in the three steps: Evaluate, Extract, and Update. After going through these three steps, it returns as output the vector V' of new parameter values and D.

3.3.1 ***Evaluate.*** The goal of the first step is to evaluate the quality of the parameter vector V used for generating the test cases T in terms of code coverage and detected bugs. To this end, SymTuner generates data d for each test case $t \in T$ (lines 3–4).

A single data d is represented by the quadruple (Br, Bu, t, V), where Br and Bu are the sets of branches covered and bugs triggered

Algorithm 3 SymTuner

Input: Test cases (T), spaces ($\mathbb{S}$), values (V), data (D), flag (*flag*)
Output: New parameter values (V') and Data (D)

1: **procedure** SymTuner($T, \mathbb{S}, V, D, flag$)
2: /* Step 1: Evaluate the quality of test cases */
3: **for each** $t \in T$ **do**
4: $D \leftarrow D \cup \{d\}$ ▷ $d = \text{Evaluate}(t, V)$
5:
6: /* Step 2: Extract effective parameter values from data D */
7: $CoreV, TotalV \leftarrow \text{Extract}(D)$
8:
9: /* Step 3: Update sampling probabilities */
10: $policy \leftarrow$ sample from $\{Exploit, Explore\}$ with prob=[α, 1-α]
11: $V' \leftarrow \langle\rangle$
12: **for** $i = 1$ **to** k **do** ▷ $\mathbb{S} = S_1 \times S_2 \times \cdots \times S_k$
13: **if** ($policy = Explore$) or ($flag = 0$) **then**
14: $(\mathcal{P}_i^c, \mathcal{P}_i) \leftarrow \text{Explore}(TotalV, S_i)$
15: **else**
16: $(\mathcal{P}_i^c, \mathcal{P}_i) \leftarrow \text{Exploit}(CoreV, TotalV, S_i)$
17: $PV_i \leftarrow \text{Sample}(S_i, \mathcal{P}_i^c, \mathcal{P}_i)$
18: $V' \leftarrow V' \cdot PV_i$ ▷ Append PV_i at the end of V'
19: **return** V', D

by the test case t, respectively. We identify a bug with an error location, a pair of the function name and the line number (e.g., (foo,3)). To obtain the two sets, Br and Bu, the Evaluate function at line 4 executes the program with each test case t and performs a coverage analysis (e.g., using gcov). The Evaluate function takes as input a single test case t and a vector V of used parameter values, and returns data d as output. SymTuner collects all data in D.

3.3.2 ***Extract.*** After SymTuner obtains the set D of data, it moves onto the next step, Extract, where it extracts as learning data two sets, *CoreV* and *TotalV*, of parameter value vectors from *CoreD* and D, respectively. Intuitively, the set *CoreD* includes only core elements of D, which we define as the smallest subset of D that covers all branches and all bugs in D. To construct *CoreD*, we first compute the set D^*:

$$D^* = \underset{D' \subseteq D}{\text{argmax}} \left| \bigcup_{(Br,Bu,\_,\_) \in D'} (Br \uplus Bu) \right|.$$

where "argmax" denotes the set of all possible arguments which maximize the given objective (e.g., the number of covered branches and detected bugs) and $\uplus$ denotes the disjoint union. From the set D^*, we define the set *CoreD* to be the smallest one in D^*, i.e., $CoreD = \text{argmin}_{D' \in D^*} |D'|$ (here we assume "argmin" returns an arbitrary single argument that minimizes the given objective). In other words, *CoreD* represents the minimum subset of D that collectively maximizes the number of covered branches and found bugs. For instance, suppose that the data D has four elements:

$$D = \{(\{b_1, b_2, b_5\}, \emptyset, \_, \_),\ (\{b_5\}, \{(foo, 3)\}, \_, \_),\\ (\{b_1, b_2, b_3, b_4\}, \emptyset, \_, \_),\ (\{b_2, b_3, b_4\}, \emptyset, \_, \_)\}$$

where each element in D consists of a quadruple of a set of covered branches, a set of bugs, a test case, and a vector of parameter values. From the set D, we can extract *CoreD* as:

$$CoreD = \{(\{b_5\}, \{(foo, 3)\}, \_, \_), (\{b_1, b_2, b_3, b_4\}, \emptyset, \_, \_)\}.$$

Calculating *CoreD* corresponds to solving the set cover problem [40], which is a well-known NP-complete problem. We compute *CoreD* using a greedy algorithm that iteratively selects the element having the largest number of uncovered branches and bugs at each stage.

From *CoreD* and *D*, we collect the learning data, *CoreV* and *TotalV*. We first obtain the set *CoreV* of effective parameter values in *CoreD* as follows:

$$CoreV = \{\!\{ V \mid (\_, \_, \_, V) \in CoreD \}\!\}.$$

Note that we deliberately define the set *CoreV* as a multiset to track the influential parameter values more effectively. For example, suppose that *CoreV* is $\{\!\{V_1, V_2, V_1\}\!\}$, where V_1 is duplicated. Since the existence of duplication implies that the duplicated value is used more than once in the set *CoreD*, we can conclude that V_1 is the vector of more influential parameter values than V_2 in terms of performance. We also collect the set *TotalV* of all parameter values used in the accumulated data *D* as:

$$TotalV = \{V \mid (\_, \_, \_, V) \in D\}.$$

Note that the set *TotalV* is a standard set which does not allow duplicated elements. The Extract function returns as output the two sets *CoreV* and *TotalV* for the final step.

3.3.3 ***Update.*** The aim of last step is to update sampling probabilities of parameter spaces based on the extracted data, *CoreV* and *TotalV*, and to generate a new vector V' of parameter values using the updated probabilities. More specifically, we update the probability functions $\mathcal{P}_i$ and $\mathcal{P}_i^c$ by using the following two policies: *Explore* and *Exploit*. In general, the exploration policy, *Explore*, gives more opportunities to parameter values which have been used less frequently. On the other hand, the exploitation policy, *Exploit*, increases the probabilities for parameter values with good performance while taking into account the number of times the values have been used. As balancing exploitation and exploration is a well-known important problem, based on trial and error, we set the sampling probability of *Exploit* and *Explore* to be 70% and 30%, respectively; that is, we set the hyper-parameter α to 0.7 at line 10 in Algorithm 3.

Exploration. When the selected policy is *Explore* or the value of *flag* is 0 (line 13), SymTuner updates the probability for the i-th parameter space S_i by using the Explore function (line 14), and then samples the i-th parameter value by using the Sample function (line 17). The Explore function takes as input the set *TotalV* of all parameter vectors used before and the i-th parameter space $S_i = (SV_i, \eta_i^{\max})$, and returns as output the updated probability functions.

To update the sampling probability, Explore($TotalV$, $(SV_i, \eta_i^{\max})$) first scores each value v' in SV_i as follows:

$$score_i(v') = \frac{1}{|\{V \in TotalV \mid v' \in V^i\}|} \tag{1}$$

where V^i denotes the i-th element of vector V. The denominator is the number of times the value v' is used as the i-th component during the symbolic execution so far. When the value v' is never used (i.e., dominator=0), we give a highest possible score for v'. The intuition is that we give higher scores to parameter vectors that have been used less frequently, so that SymTuner explores unseen parameter values. With $score_i$, we define the probability function $\mathcal{P}_i : SV_i \to [0, 1]$ as follows:

$$\mathcal{P}_i(v') = \frac{score_i(v')}{\sum_{sv \in SV_i} score_i(sv)} \tag{2}$$

The probability is the normalized score of v' divided by the sum of the scores of all parameter values in SV_i. Intuitively, if the number of all distinct parameter values used is the same, the sampling probability is evenly distributed; SymTuner samples the parameter values at complete random in this case.

Likewise, to obtain the probability $\mathcal{P}_i^c$ of the number of sampling in the i-th space $S_i = (SV_i, \eta_i^{\max})$, we calculate the score for each number m' of sampling ($1 \le m' \le \eta_i^{\max}$) using $score_i^c$ defined as:

$$score_i^c(m') = \frac{1}{|\{V \in TotalV \mid m' = |V^i|\}|} \tag{3}$$

The denominator is the number of times the value m' equals to the sample size ($|V_i|$). We compute $\mathcal{P}_i^c : [1, \eta_i^{\max}] \to [0, 1]$ as follows:

$$\mathcal{P}_i^c(m') = \frac{score_i^c(m')}{\sum_{1 \le m \le \eta_i^{\max}} score_i^c(m)} \tag{4}$$

At lines 17–18, with the two updated probabilities $\mathcal{P}_i$ and $\mathcal{P}_i^c$, the algorithm generates new i-th parameter value PV_i using the Sample function, and then adds PV_i to the vector V'.

Exploitation. Besides exploration, we employ an exploitation policy (*Exploit*) to learn the sampling probabilities of each parameter space (lines 16). The policy uses the Exploit function which increases the sampling probability of the values that have been used more often as influential parameter values in *CoreV*.

Basically, the exploitation method computes the probability $\mathcal{P}_i$ of each parameter space in the same way as the exploration method. The scoring function, however, is different and defined as follows:

$$score_i(v') = \frac{|\{V \in CoreV \mid v' \in V^i\}|}{|\{V \in TotalV \mid v' \in V^i\}|}$$

Intuitively, the score for the value v' indicates how often v' is used as influential parameter values in *CoreV*. More precisely, the numerator represents the number of times the value is used in the i-th component in the set *CoreV*. Note that we divide this number by the total number of times the value has been used, preferring parameter values with higher "hit rates" instead of just preferring parameters in *CoreV* simply because they have been tried frequently. Similarly, we define the score function $score_i^c$ for $\mathcal{P}_i^c$ as follows:

$$score_i^c(v') = \frac{|\{V \in CoreV \mid v' = |V^i|\}|}{|\{V \in TotalV \mid v' = |V^i|\}|}$$

After the score calculation, the *Exploit* policy updates the probabilities $\mathcal{P}_i$ and $\mathcal{P}_i^c$ using the equations (2) and (4) at line 16. Then, it generates new i-th parameter value based on the Sample function.

Through the three steps, Evaluate, Extract, and Update, Algorithm 3 accumulates the set *D* of learning data, and returns it as output. By repeating Algorithm 3, the set *D* is continuously updated, and SymTuner gradually makes a smart decision on how to sample k parameter values from k predefined parameter spaces towards maximizing the performance of symbolic execution.

4 EXPERIMENTS

In this section, we experimentally evaluate the effectiveness of our approach. Research questions are as follows:

(1) **Coverage:** How effectively does SymTuner enhance symbolic execution in terms of branch coverage?
(2) **Bug-finding:** Does the interaction between SymTuner and symbolic executor enhance the bug-finding ability?
(3) **Impact of parameters and the spaces:** What is the most influential parameter? How does the performance of our approach change depending on different parameter spaces?
(4) **Generality:** Is SymTuner applicable to concolic testing, another approach to dynamic symbolic execution?

We used KLEE [11][2] as a base symbolic executor to interact with SymTuner because KLEE is one of the most popular and actively maintained symbolic execution tools available today. All experiments were conducted on a machine with two Intel Xeon Gold 6230R and 256GB RAM.

4.1 Experimental Settings

4.1.1 **Predefined Parameter Spaces.** SymTuner takes as input predefined parameter spaces ($\mathbb{S}$). In our experiments, we aimed to tune all, more precisely 20, parameters in Figure 1, where their types consist of 7 integer, 3 double, 4 string, and 6 boolean types. In particular, for the first 14 parameters which are not boolean types, we defined their spaces as:

$$
\begin{aligned}
&S_{search} = (\{s_1, s_2, \cdots, s_{10}\}, 1),\\
&S_{args} = (\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, 5),\\
&S_{files} = S_{stdin} = (\{4, 8, 12, 16, 20\}, 1),\\
&S_{mem} = (\{500, 1000, 1500, 2000, 2500\}, 1),\\
&S_{batch} = (\{6000, 8000, 10000, 12000, 14000\}, 1),\\
&S_{instr\_time} = (\{10, 20, 30, 40, 50\}, 1),\\
&S_{array\_size} = (\{3000, 3500, 4000, 4500, 5000\}, 1),\\
&S_{fork} = S_{cpfork} = S_{solve} = (\{0.25, 0.5, 1, 2, 4\}, 1),\\
&S_{switch} = (\{\text{“}simple\text{”}, \text{“}internal\text{”}\}, 1), S_{seed} = (\{\}, 1),\\
&S_{external} = (\{\text{“}concrete\text{”}, \text{“}all\text{”}\}, 1),
\end{aligned}
$$

The rationale behind the spaces is twofold. First, for each parameter, we simply chose 5–10 values around the value used in Figure 1. Second, we tried to subsume the settings used in prior work [11, 45, 63, 66]. In the first space (S_{search}), $s_1, \ldots, s_{10}$ denote the ten search strategies implemented in KLEE. Note that we also tuned the seed input parameter ('--seed-file') not involved in Figure 1 as its impact on the performance of symbolic execution is well-known in the literature [14, 19, 37, 54]. Despite its importance, however, it is not appropriate to predefine the seed sample space (e.g., S_{seed}) because the corpus of seed inputs highly depends on the program under test, which requires additional manual efforts for the end-users to use SymTuner. Thus, we did not predefine S_{seed}, but let it be dynamically determined during symbolic execution. More precisely, on every iteration of the loop at lines 5–8 in Algorithm 2, SymTuner calculates top-20 test cases in terms of covered branches and detected bugs, and updates S_{seed} with them. That is, unlike other predefined spaces, the candidate values in S_{seed} may change as the learning progresses.

[2]We used KLEE-2.0 released in March 2019.

Table 1: 12 benchmark programs

Programs	LOC	# of Branches	Programs	LOC	# of Branches
xorriso-1.5.2	161K	49,162	enscript-1.6.6	49K	3,796
gcal-4.1	89K	15,799	combine-0.4.0	32K	2,357
grep-3.4	82K	8,225	trueprint-5.4	12K	2,518
gawk-5.1.0	77K	10,720	diff (diffutils-3.7)	9K	7,612
sed-4.8	66K	6,798	du (coreutils-8.32)	8K	6,653
nano-4.9	54K	10,436	ls (coreutils-8.32)	5K	3,776

In total, the product of the 20 parameter spaces ($\mathbb{S}$) induces 10^{16} different parameter settings.

4.1.2 **Baselines.** We compared our approach (Klee+SymTuner) with three baselines: $\text{Klee}_{\text{default}}$, $\text{Klee}_{\text{hand}}$, and Klee+RandTuner. The first baseline, $\text{Klee}_{\text{default}}$, uses the default parameter values provided by KLEE without any modification. The second one, $\text{Klee}_{\text{hand}}$, uses the hand-tuned parameter setting in Figure 1. More precisely, its configuration is the same as the parameter values provided in the KLEE documentation [49], which has been a conventional choice in prior work [17, 18, 23, 50, 63]. The last one, Klee+RandTuner, is a baseline that randomly samples parameter values from our parameter spaces defined in Section 4.1.1; we simply substituted RandTuner for SymTuner on line 6 in Algorithm 2.

For a fair comparison of $\text{Klee}_{\text{default}}$ and $\text{Klee}_{\text{hand}}$, we ran each baseline in two different modes, respectively, and then reported the best results. The only difference between the two modes is in how the given time budget is used. The first method is to perform symbolic execution (Algorithm 1) only once for the total budget (e.g., 10h) while the other is to run Algorithm 1 multiple times by dividing the total budget into smaller budgets. More precisely, the second method is to run Algorithm 2 without the parameter-tuning process.

4.1.3 **Benchmarks and Time Budgets.** We used 12 GNU open-source C programs in Table 1. Our benchmark suite includes the largest programs used in prior works [14, 16, 17, 47, 54]. For example, the last three programs in Table 1 are the largest (or second largest) ones in GNU coreutils-8.32 and diffutils-3.7. For each benchmark program, we collected the most recent releases (as of March 2020). For all experiments, we set the time budget to 10 hours for each benchmark program. We repeated each experiment 4 times, and reported average results.

4.2 Branch Coverage

Our approach significantly outperformed the three baselines on all benchmarks in terms of branch coverage. On average over all benchmark programs, Klee+SymTuner achieved 31% and 56% higher branch coverage than $\text{Klee}_{\text{hand}}$ and $\text{Klee}_{\text{default}}$, respectively. SymTuner also succeeded in covering 12% more branches than RandTuner, showing the true benefit of online learning algorithm.

As the final outputs of our approach (Algorithm 2) and three baselines are the test cases generated during symbolic execution, we depicted the coverage graph over time in Figure 3 by accumulating the number of branches covered by the test case generated at each time step. To do so, we used gcov, a tool for measuring code coverage.

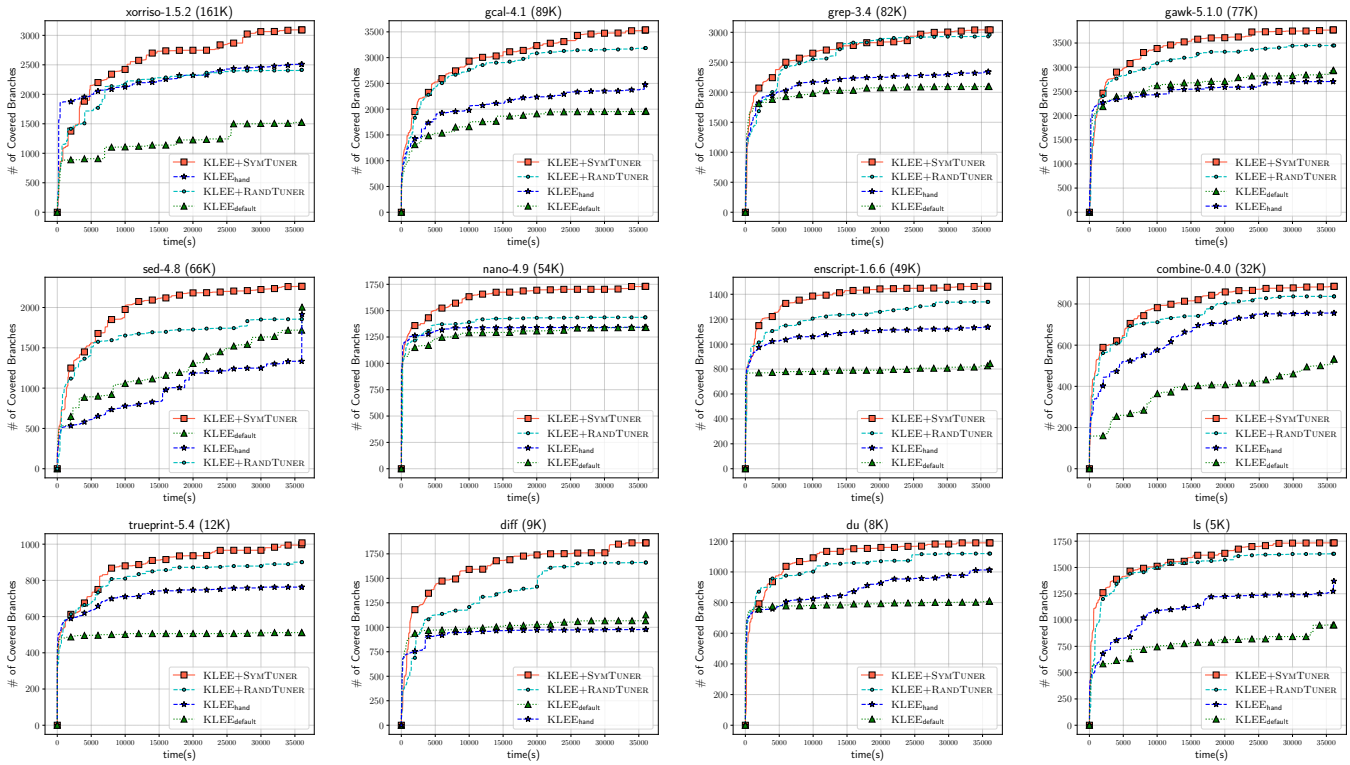

Figure 3: The average number of covered branches achieved by our approach and three baselines on 12 benchmarks

The results in Figure 3 show that Klee+SymTuner consistently achieves the highest branch coverage on all benchmarks. In particular, the results for the two largest benchmarks, `xorriso` and `gcal`, are noteworthy; Klee+SymTuner was able to cover 3,093 branches on average for `xorriso` while $\text{Klee}_{\text{hand}}$ and Klee+RandTuner covered 2,509 and 2,415 branches, respectively. For `gcal`, the average number of branches covered by Klee+SymTuner was 3,539, which is 353 and 1,062 more than Klee+RandTuner and $\text{Klee}_{\text{hand}}$, respectively.

Excluding our approach, Klee+RandTuner was generally better than the other two baselines, where this result implies that performing symbolic execution with various parameter values is usually more effective than running it with the fixed values. Among $\text{Klee}_{\text{hand}}$ and $\text{Klee}_{\text{default}}$, the former achieved 19% higher branch coverage than the latter on all benchmarks. That is, using hand-tuned parameter values was better than blindly using the default values provided in KLEE.

One interesting point is that Klee+RandTuner is sometimes even inferior to the two baselines, $\text{Klee}_{\text{hand}}$ and $\text{Klee}_{\text{default}}$, which do not change the parameter values at all during symbolic execution. On `xorriso` and `sed`, Klee+RandTuner managed to cover about 100 and 150 branches less than $\text{Klee}_{\text{hand}}$ and $\text{Klee}_{\text{default}}$, respectively; Klee+RandTuner achieved the lowest coverage on `sed`. The instability of RandTuner supports that our approach (Algorithm 2) is essential to consistently achieve higher coverage.

The standard deviations of branch coverage averaged over all benchmarks and trials are as: SymTuner(122), RandTuner(99), $\text{Klee}_{\text{default}}$(102), and $\text{Klee}_{\text{hand}}$(60); these differences are insignificant considering the coverage gap between ours and the baselines.

Table 2: The branch coverage achieved by running SymTuner and RandTuner with multiple cores in parallel

	# of Cores	2	4	6	8	**10**
xorriso-1.5.4	SymTuner	3,260	3,385	3,604	4,083	**5,202**
	RandTuner	2,726	3,225	3,440	3,441	3,631
gcal-4.1	SymTuner	4,243	4,397	4,538	4,561	**4,757**
	RandTuner	3,825	4,131	4,221	4,256	4,315

Additionally, we investigated whether SymTuner still outperforms RandTuner even when running them in parallel. We wondered if running KLEE with more diverse parameters simply by using many cores in parallel would diminish the advantage of SymTuner compared to RandTuner. So, we compared SymTuner and RandTuner by accumulating the results of running them (e.g., Algorithm 2) with different number of cores in parallel, respectively. Table 2 reports the number of covered branches achieved by each technique according to the number of cores used in parallel on the two largest benchmarks: `xorriso` and `gcal`. The results show that the difference in branch coverage between the two techniques becomes larger when more cores are used. For instance, on `xorriso`, running SymTuner with 10 cores in parallel succeeded in covering about 1,500 more branches than running RandTuner with the same settings. That is, even in parallel settings, smartly tuning

Table 3: Comparison of bug-finding ability of the three baselines and SymTuner. (SymTuner = Klee+SymTuner)

Benchmarks	Error-Types	Error Locations	Bug-Triggering Test Cases	SymTuner	RandTuner	$\text{Klee}_{\text{hand}}$	$\text{Klee}_{\text{default}}$
gcal-4.1	Segmentation fault	'Line: 740 in /src/file-io.c'	`"@/⊙"` (⊙ denotes an ASCII character of 1.)	✔	✔	✔	✔
	Segmentation fault	'Line: 3956 in /src/gcal.c'	`"@/" "--at=5"`	✔	✘	✘	✘
	Segmentation fault	'Line: 72 in /libc/string/strncasecmp.c'	`"@.../" "@.." "@../" "--u=Z="`	✔	✘	✘	✘
	Abnormal termination	'Line: 27 in /libc/string/strcpy.c'	`"/#" "⊙" A` (A denotes a symbolic file.)	✔	✘	✔	✘
	Abnormal termination	'Line: 29 in /libc/string/memcpy.c'	`"@/" "-#__⊙" A`	✔	✔	✘	✔
enscript-1.6.6	Segmentation fault	'Line: 1880 in /libc/stdio/_vfprintf.c'	`"--to" "" "" "" ""`	✔	✔	✘	✘
gawk-5.1.0	Abnormal termination	'Line: 1337 in main.c'	`"-W" "nost"`	✔	✔	✘	✘
combine-0.4.0	Segmentation fault	'Line: 385 in /src/field.c'	`"-f" "--field=,"`	✔	✔	✔	✔
	Segmentation fault	'Line: 458 in /src/field.c'	`"--re" "" "--fi" "d.e0-2,"`	✔	✘	✘	✘
	Segmentation fault	'Line: 633 in /src/df_options.c'	`"-Pp" "--no-ch" "--fi" "r.o1" "--r="`	✔	✔	✘	✘
	Memory-exhaustion	'Line: 48 in /libc/string/memmove.c'	`"-ecut" "--fiel" "8,--1"`	✔	✘	✘	✘

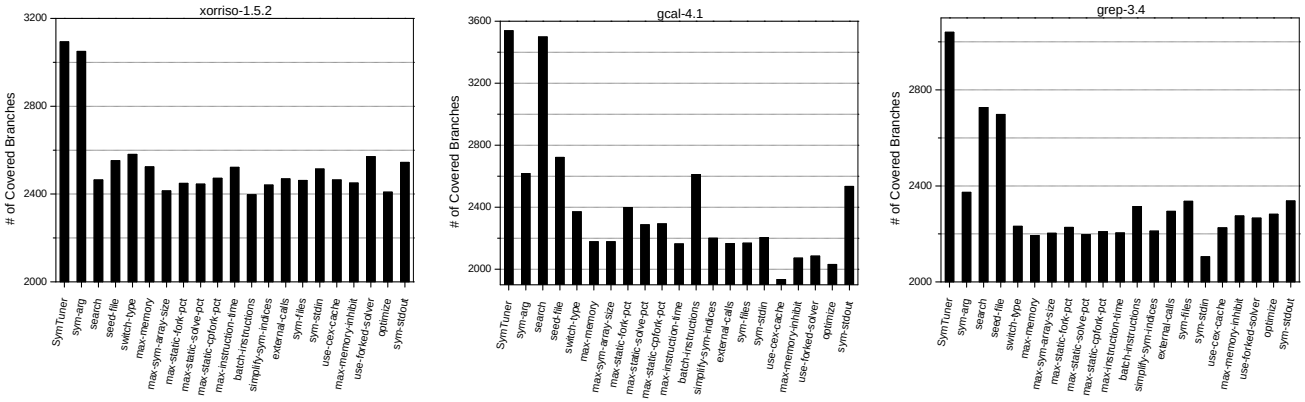

Figure 4: The average branch coverage achieved by tuning only individual parameter on the three largest benchmarks

parameters is still more effective than trying various parameters indiscriminately. As future work, to further improve the effectiveness of SymTuner when running it in parallel, we plan to apply the core ideas of the existing techniques such as swarm testing [26, 31].

4.3 Bug-finding

Table 3 shows that SymTuner also has considerable promise in improving the bug-finding ability of KLEE. In summary, SymTuner detected 11 different real-bugs from four open-source programs while the best one among three baselines, RandTuner, found just six of them.

Columns in Table 3 denote the benchmark program, error-type, error-location, bug-triggering test case produced by SymTuner, and indication of success (✔) or failure (✘) for each technique. In particular, we note that the failure mark '✘' indicates that the corresponding technique completely failed to find the bug within 40 hours (10h × 4 repetitions). Conversely, if succeeding on the bug detection at least once during the four trials, we marked the result as 'success' (✔).

Our approach (Klee+SymTuner) found 11 different bugs in total, and we classified them into three error-types: abnormal termination, segmentation fault, and memory exhaustion. The first two error-types cause the program to crash while the third one is a performance bug. For example, the bug-triggering input (`"@/" "--at=5"`) generated by Klee+SymTuner for the program `gcal` causes a segmentation fault which terminates the program abnormally. Klee+SymTuner also found fatal bugs in `combine`; the input (`"-ecut" "--fiel" "8,--1"`) leads to a serious performance degradation which consumes all available memory of the machine. These bug-triggering test cases in Table 3 are easily reproducible. For example, on gawk-5.1.0, executing the command (`./gawk "-W" "nost"`) will abort the program execution immediately. An unexpected result in Table 3 is that RandTuner failed to find a bug that $\text{Klee}_{\text{hand}}$ discovered in `gcal`; that is, RandTuner is unstable even in terms of bug-finding capability.

Additionally, we also investigated whether RandTuner could find more bugs when running it in parallel with multiple CPU cores. Compared to running RandTuner on a single core, using 10 cores in parallel was able to find more bugs, but it still failed to find some bugs which were discovered by SymTuner (e.g., the bug found in the file 'strncasecmp.c' of `gcal`). When we executed RandTuner for much longer (e.g., 20h) using 10 cores in parallel, RandTuner was eventually able to find all bugs that SymTuner found with a single core for 10 hours.

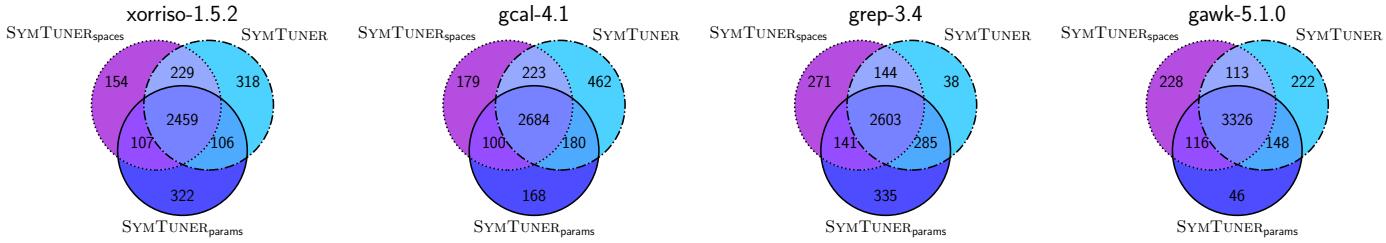

Figure 5: Venn-diagrams illustrating the sets of branches covered by SymTuner with different parameter spaces

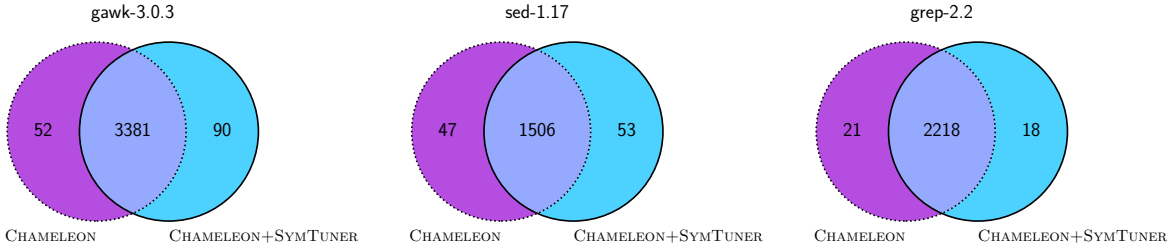

Figure 6: Venn-diagrams depicting the sets of covered branches by Chameleon with/without SymTuner

4.4 Impact of Parameters and their Spaces

Impact of Individual Parameters. We investigated which of the 20 parameters in Table 1 had the greatest impact on the performance of symbolic execution for the three largest benchmarks, `xorriso`, `gcal`, and `grep`. To do so, we performed symbolic execution while tuning each parameter one by one, and reported the average branch coverage for each parameter-tuning with the same setting of SymTuner (10h × 4 repetitions).

Figure 4 shows that the most influential parameters tend to be similar across the three programs, but the least influential ones are different depending on the target program. The two parameters, symbolic command-line arguments ('--sym-arg') and seed input ('--seed-file'), were consistently included in the top-4 most influential parameters for all the benchmarks. Also, the most crucial parameter for `gcal` and `grep` was equal as search strategy ('--search'), but the search strategy parameter was not included even in top-10 important parameters for `xorriso`, which means that every important parameter was not shared across all the programs. Likewise, the forth most important parameter ('--batch-instructions') on `gcal` was the most unimportant one on `xorriso`. On the other hand, the least influential parameter for each program is different as: '--batch-instructions' (`xorriso`), '--use-cex-cache' (`gcal`), and '--sym-stdin' (`grep`). These results support our claim that we should take a program-adaptive method to tune various parameters for symbolic execution.

An unexpected observation from Figure 4 is that the parameter type itself seems to be related to its importance. First, the string-type parameters (e.g., '--search', '--seed-file') ranked the most in the most important top-4 parameters while none of the parameters belonged to the least important top-4 parameters; that is, the string-type parameter is important to tune carefully. Second, boolean type parameters are less valuable to tune than the other type parameters as boolean types account for 50% of the least important top-4 parameters.

Table 4: 14 Parameter spaces added for SymTuner$_{\text{Params}}$

Boolean	Boolean	Integer or (Double)
use-branch-cache	cex-cache-exp	redzone-size ({5, 10, 15, 20, 25, 30},1)
use-constant-arrays	cex-cache-superset	seed-time ({5, 10, 15, 20, 25, 30},1)
solver-optimize-divides	cex-cache-try-all	max-stack-frames ({6000, 7000, 8000, 9000, 10000},1)
allocate-determ	equality-substitution	allocate-determ-size ({50, 100, 150, 200, 250},1)
rewrite-equalities		max-static-cpsolve-pct ({0.25, 0.5, 1, 2, 4},1)

The results in Figure 4 also demonstrate that tuning only the most influential parameter is less beneficial than SymTuner which adjusts the 20 parameters simultaneously. In particular, SymTuner covered 11.5% more branches on `grep` than tuning only the most influential parameter ('--search'). Also, in terms of bug-finding capability, we found that tuning only the most influential parameter on `gcal` was able to discover only a single bug located in '/src/file-io.c' among the total 5 bugs found by SymTuner on `gcal` in Table 3.

Impact of Parameter Spaces. We evaluated how the performance of SymTuner changes depending on different parameter spaces. To do so, we compared SymTuner with its two variants, SymTuner$_{\text{Spaces}}$ and SymTuner$_{\text{Params}}$, having different parameter spaces. The former (SymTuner$_{\text{Spaces}}$) is a variant that doubles each space of the 10 parameters with integer or double type defined in Section 4.1.1, respectively. For example, the space of S_{args} of SymTuner is between 1 and 10, but the space for SymTuner$_{\text{Spaces}}$ will be between 1 and 20. That is, the parameter spaces for the first variant will be $2^{10} * 10^{16}$, which is about 1,000 times larger than the spaces for SymTuner. The latter (SymTuner$_{\text{Params}}$) is another variant that tunes more parameters than the 20 parameters to be

tuned originally in SymTuner. More specifically, this variant aimed to simultaneously tune a total of 34 parameters, including the 20 parameters that SymTuner tuned and the 14 additional parameters in Table 4; we manually added the 14 parameters that are likely to affect the performance of symbolic execution among the total parameters provided to KLEE. We evaluated the two variants with the same settings (e.g, 10h × 4 repetitions) as SymTuner on 4 largest benchmarks, and reported the average results.

Figure 5 shows the Venn-diagrams which describe the relationships in terms of the sets of branches reached by each technique. The results show that SymTuner is able to cover different code areas of the target program effectively depending on the parameter spaces to be tuned. In terms of the total number of covered branches, SymTuner achieved the highest branch coverage on `gcal` and the lowest coverage on `grep`. Exactly opposite, $\text{SymTuner}_{\text{Params}}$ achieved the lowest branch coverage on `gcal` and the highest coverage on `grep`. Figure 5 also shows that there exist many branches that SymTuner and its two variants are able to exclusively reach. For example, $\text{SymTuner}_{\text{Spaces}}$ exclusively covered 271 branches on `grep`, and $\text{SymTuner}_{\text{Params}}$ succeeded in covering 332 unique branches on `xorriso`. That is, the potential of SymTuner may vary depending on different parameter spaces.

4.5 Generality of SymTuner

We checked if our approach is applicable to concolic testing [25, 53], another major approach to dynamic symbolic execution. We applied SymTuner to CREST [21] as it is a publicly available tool and Chameleon [16], the state-of-the-art technique for tuning search heuristics for concolic testing, is implemented on top of CREST [58]. Hence, our approach (Chameleon+SymTuner) aims to tune the other parameters while letting Chameleon tune search strategies in its own way. To do so, we implemented SymTuner on top of Chameleon, and figured out whether SymTuner was able to enhance Chameleon.

Unlike KLEE, CREST only provides three external parameters; others are hard-coded inside the tool and difficult to tune from the outside. In our experiments, we tried to tune all of the three parameters: symbolic command-line argument, seed input, and the number of program executions. On the basis of the parameter values used in Chameleon [58], we defined their spaces as follows:

$$S_{args} = (\{12, 14, 16, 18, 20, 22, 24, 26, 28, 30\}, 1),$$
$$S_{seed} = (\{I_0\}, 1),$$
$$S_{execution} = (\{3000, 3500, 4000, 4500, 5000\}, 1)$$

In particular, we first initialized the space S_{seed} with an initial input (I_0) provided in Chameleon [58], and let it be dynamically decided during concolic testing like the space S_{seed} in Section 4.1.1. For evaluation, we used the same three benchmark programs taken from Chameleon, allocated the time budget to 10 hours, and reported the number of covered branches averaged over 5 times.

Figure 6 shows that our approach (Chameleon+SymTuner) has its own benefit in terms of exclusively covered branches. For the three benchmarks, Chameleon+SymTuner succeeded in covering about 30% more unique branches than Chameleon alone. We expect that the usefulness of SymTuner will be greater if various parameters provided in KLEE are also added to CREST in the future. Note that since SymTuner uses a symbolic execution tool (e.g., KLEE and CREST) as a blackbox, we expect that applying SymTuner to other symbolic execution tools [51, 56] does not require much effort.

4.6 Threats to Validity

(1) We evaluated SymTuner only for KLEE and CREST. We chose them as they are the representative symbolic executors for C programs, but the results reported in this paper may not be valid for other testing tools such as EVOSUITE [24], a widely used unit testing tool. (2) We manually defined the 20 parameter spaces of KLEE by choosing 5–10 values around the value used in Figure 1. However, these predefined spaces may not be appropriate for the other target programs beyond our 12 benchmarks. (3) Our approach (Algorithm 2) involves hyper-parameters, e.g., η_{ratio} and η_{step}, which were selected heuristically. These values may need to be set properly for target programs. (4) We used 12 programs including the largest real-world programs (up to 161KLoC) among those used in prior works [14, 16, 17, 47, 54] for evaluating KLEE. However, these might not be representative enough.

5 RELATED WORK

Improving Symbolic Execution. To our knowledge, SymTuner is the first technique to tune general parameters of symbolic execution automatically. Over the past decade, a lot of research has been conducted to advance symbolic execution, and they can be classified into three groups according to the main approach: search strategies [10, 45, 54, 62], pruning techniques [6, 9, 34, 66], and constraint solving techniques [23, 35, 52, 65]. First, prior works on search strategies aim to preferentially explore the execution paths of the program that are likely to maximize the performance (e.g., code coverage). For example, the CFDS strategy [10] prioritizes the program's paths closest to the branches that have not yet been reached, and the CGS strategy [54] favors exploring the paths with a new context (i.e., new sequence of branches). Second, path-pruning techniques focus on removing the redundant paths of the program based on the predefined criteria. For instance, Jaffar et al. [34] presented a criterion that eliminates the execution paths guaranteed not to reach the error locations. Lastly, diverse techniques have emerged to reduce the cost for constraint solving, one major bottleneck in symbolic execution, by simplifying the array constraints [52] or reusing the constraint solutions [35, 65]. SymTuner is orthogonal to the above three approaches and we believe that SymTuner can further enhance the existing approaches by automatically tuning external parameters.

Software Testing with Learning. Our approach follows a recent trend in software testing that leverages machine learning [14–18, 42, 44, 55, 57]. ParaDySE [14] boosts concolic testing by automatically generating search strategy via offline learning. Using online learning technique, Chameleon [16] adaptively switches the search strategies of concolic testing. LEO [18] aims to improve the efficacy of symbolic execution by learning how to use compiler optimizations for code transformation. In Android GUI testing, QBE [42] uses reinforcement learning to explore GUI actions that are likely to detect bugs and increase activity coverage. RETECS [57] learns how to preferentially select buggy test cases in Continuous Integration

based on reinforcement learning. In this paper, we use learning for a novel application, i.e., tuning symbolic execution parameters.

Search-based Software Testing. Our work can be considered an instance of search-based software testing (SBST) [1, 4, 27, 46] using meta-heuristic search technique in the field of search-based software engineering [3, 28–30]. SBST aims to find good solutions from an extremely large search space in a reasonable time for enhancing testing efficacy. To do so, each technique in SBST defines its own optimization problem and proposes a fitness function specialized for solving the problem. In our work, we formulated the problem of tuning parameter values of symbolic execution as an optimization problem that maximizes both the number of covered branches and found bugs, and presented a specialized algorithm to solve it.

Automatic Parameter Tuning. Automatic parameter tuning has been studied extensively in various fields. For example, researchers have developed domain-specific algorithms for database systems [22, 61], web systems [8, 64], image segmentation [59], and big data processing systems [20, 43]. Our work lies in this line of research and presents an algorithm specialized for symbolic execution. Existing frameworks for algorithm configuration (e.g., ParamILS [33], OpenTuner [2]) are inappropriate for our purpose. Note that these are offline approaches; they aim to discover good parameter settings of algorithms and the same settings are used without change at runtime. By contrast, the main benefit of SymTuner comes from adaptively adjusting the parameter values online (during symbolic execution), which is crucial in our case as optimal parameter values vary significantly depending on the target programs (Section 4.4). Also, using these tools effectively often requires domain expertise [2]; our goal is to enable users to use symbolic execution without any prior knowledge.

6 CONCLUSION

Automatic tuning of symbolic execution parameters has received little attention despite its importance in practice. In this paper, we called for attention to this problem and presented SymTuner for automatically tuning parameters of symbolic execution via online learning. Experimental results showed that running KLEE in concert with SymTuner leads to sharp increases in branch coverage and found bugs. We hope that SymTuner will help end-users to maximally benefit from powerful yet difficult-to-use modern symbolic execution tools.

ACKNOWLEDGMENTS

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-51. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair) and the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program (IITP-2022-2020-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation). This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A5A1021944, No.NRF-2021R1C1C2006410).

REFERENCES

[1] Wasif Afzal, Richard Torkar, and Robert Feldt. 2009. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology* (2009), 957–976.

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT'14)*. 303–315.

[3] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* (2013), 594–623.

[4] Andrea Arcuri and Xin Yao. 2008. Search based software testing of object-oriented containers. *Information Sciences* (2008), 3075–3095.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. https://doi.org/10.1145/3182657

[6] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. 351–366.

[7] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT–a Formal System for Testing and Debugging Programs by Symbolic Execution. *SIGPLAN Not.* (1975), 234–245.

[8] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. 2009. A Reinforcement Learning Approach to Online Web Systems Auto-Configuration. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS '09)*. 2–11.

[9] Suhabe Bugrara and Dawson Engler. 2013. Redundant State Detection for Dynamic Symbolic Execution. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. 199–212.

[10] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. 443–446.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. 209–224.

[12] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*. 1066–1071.

[13] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

[14] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 1244–1254.

[15] Sooyoung Cha, Seonho Lee, and Hakjoo Oh. 2018. Template-guided Concolic Testing via Online Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. 408–418.

[16] Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. 235–245.

[17] Sooyoung Cha and Hakjoo Oh. 2020. Making Symbolic Execution Promising by Learning Aggressive State-Pruning Strategy. In *The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*.

[18] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP '18)*.

[19] 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 *2020 IEEE Symposium on Security and Privacy (S&P '20)*. 1580–1596.

[20] Dazhao Cheng, Jia Rao, Yanfei Guo, and Xiaobo Zhou. 2014. Improving MapReduce Performance in Heterogeneous Environments with Adaptive Task Tuning. In *Proceedings of the 15th International Middleware Conference (Middleware '14)*. 97–108.

[21] CREST. A concolic test generation tool for C. 2008. https://github.com/jburnim/crest.

[22] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with ITuned. *Proc. VLDB Endow.* (2009), 1246––1257.

[23] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. 2018. PARTI: A Multi-Interval Theory Solver for Symbolic Execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. 430–440.

[24] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. 416–419.

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 213–223.

[26] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*. 78–88.

[27] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST '15)*. 1–12.

[28] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* (2001), 833–839.

[29] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* (2012), 1–61.

[30] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. 2010. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical software engineering and verification*. 1–59.

[31] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. 2011. Swarm Verification Techniques. *IEEE Transactions on Software Engineering* (2011), 845–857.

[32] W. E. Howden. 1977. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering* (1977), 266–278.

[33] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* (2009), 267–306.

[34] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. 2013. Boosting Concolic Testing via Interpolation. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. 48–58.

[35] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*. 177–187.

[36] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. 2015. Learning Combinatorial Interaction Test Generation Strategies Using Hyperheuristic Search. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. 540–550.

[37] Timotej Kapus, Frank Busse, and Cristian Cadar. 2020. Pending Constraints in Symbolic Execution for Better Exploration and Seeding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. 115––126.

[38] Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 774–784. https://doi.org/10.1145/3338906.3338936

[39] Timotej Kapus, Martin Nowack, and Cristian Cadar. 2019. Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?. In *Tests and Proofs*, Dirk Beyer and Chantal Keller (Eds.). Springer International Publishing, Cham, 41–54.

[40] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. 85–103.

[41] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[42] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST '18)*. 105–115.

[43] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. 2014. MRONLINE: MapReduce Online Performance Tuning. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '14)*. 165–176.

[44] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. 554–559.

[45] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications (OOPSLA '13)*. 19–32.

[46] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 153–163.

[47] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic Execution with Existential Second-Order Constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. 389–399.

[48] M. Nowack. 2019. Fine-Grain Memory Object Representation in Symbolic Execution. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 912–923.

[49] OSDI'08_Coreutil_Experiments. 2008. https://klee.github.io/docs/coreutils-experiments.

[50] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred Concretization in Symbolic Execution via Fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. 228–238.

[51] Corina S Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10)*. 179–180.

[52] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating Array Constraints in Symbolic Execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. 68–78.

[53] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*. 263–272.

[54] Hyunmin Seo and Sunghun Kim. 2014. How We Get There: A Context-guided Search Strategy in Concolic Testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. 413–424.

[55] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints.. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS '19)*.

[56] Jiri Slaby, Jan Strejček, and Marek Trtík. 2013. Symbiotic: synergy of instrumentation, slicing, and symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*. 630–632.

[57] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. 12–22.

[58] Chameleon. A tool that performs concolic testing with adaptively changing search heuristics. 2019. https://github.com/kupl/chameleon.

[59] Thomas Torsney-Weir, Ahmed Saad, Torsten Moller, Hans-Christian Hege, Britta Weber, Jean-Marc Verbavatz, and Steven Bergner. 2011. Tuner: Principled parameter finding for image segmentation algorithms using visual response surface exploration. *IEEE Transactions on Visualization and Computer Graphics* (2011), 1892–1901.

[60] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 350–360.

[61] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024.

[62] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards Optimal Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 291–302.

[63] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan. 2015. DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE '15)*. 620–631.

[64] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. 2004. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. 287–296.

[65] Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*. 144–154.

[66] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2018. Eliminating Path Redundancy via Postconditioned Symbolic Execution. *IEEE Transactions on Software Engineering* (2018), 25–43.