Perspectives on Search Strategies in Automated Test Input Generation (Extended Version)

Yang CAO, Yanyan JIANG, Chang XU, Jun MA, Xiaoxing MA

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China

Abstract Automatically generating high-quality test inputs is a central problem in software testing and maintenance. Although being extensively studied, there is still a large gap between two main-stream approaches, which are believed to be intrinsically different: fuzz testing (fuzzing) and dynamic symbolic execution (DSE). This paper tries to shed some light on bridging the gap by presenting a unified perspective to study existing automated test input generation techniques as a heuristic search procedure. This paper qualitatively studied existing pure fuzzing and DSE techniques, and empirically compared two representative techniques AFLfast and KLEE by examining covered and non-covered code on real-world programs. Based on these insights, we study hybrid techniques of fuzzing and DSE, and discuss findings and implications on the problem of test input generation.

Keywords automated test input generation, fuzzing, dynamic symbolic execution

1 Introduction

Automated generation of test inputs has been a central issue in software testing research [1]. With astronomically large search spaces of test inputs, one has to afford to test a program with a *subset* of inputs [2]. Therefore, the generation of a limited number of *high-quality* inputs to reveal bugs, crashes, and hangs becomes a key challenge [2].

The two most extensively studied approaches to automated test input generation are *fuzzing* [3] and *dynamic symbolic execution* (DSE) [4–6]. Fuzzers [7] feed the program under test with a large amount of invalid, unexpected, or random data as inputs to trigger unintended program behaviors. Modern fuzzers are usually built on heuristic algorithms (e.g., genetic algorithm [8]) and are usually guided by the profiling data over executions (e.g., coverage information [9]). DSE takes a different approach to the input space exploration. DSE collects path constraints over symbolic variables along with a program execution, and later solves these constraints for new test inputs that are guaranteed to have distinct future execution paths. Though constraint solving is an intractable problem, DSE under proper treatments [10] can effectively find bugs in large-scale real-world programs.

Both fuzzing and DSE can effectively generate structural test inputs for non-trivial programs without human aids, and are extensively studied in existing literatures [5, 6, 11–13]. However, they are also *considerably different*. To further understand in what sense they are similar or different, we raise the following question:

(**Q1**) How do existing fuzzing and DSE techniques model the input space and manage the search procedure?

Furthermore, to understand the strengths and limitations of both techniques, a qualitative study on the search procedure is insufficient. We are particularly interested in the actual performance of fuzzing and DSE on practical programs. Quantitatively understanding the limitations of existing techniques is crucial to the development of future ones. Therefore, we raise the follow-up question:

(**Q2**) What portions of code are difficult to reach in the search space for the existing techniques?

This paper tires to answer these questions and shed some light on how to design more effective test input generation techniques (fuzzing, DSE or their hybrids). This paper makes the following major contributions:

- A *unified perspective* to model and characterize the test input generation problem and its solutions (fuzzing, DSE, and potential future techniques). A mini survey (Section 4) under the unified perspective (Section 3) answers Q1.
- An *empirical study* to investigate the distinctive power and limitations of fuzzing and DSE when applied to real-world programs. The empirical study in Section 5 answers Q2.

A new perspective. We found that an automated test input generation technique can be characterized by a three-element

tuple $\langle N, S, H_0 \rangle$:

1. The *neighborhood definition* function $N(H)$ defines the "adjacent" test inputs in the search space of H;
2. The *neighborhood selection* function $S(N(H))$ defines the actual search strategy that prioritizes, prunes, and finally generates test inputs in $N(H)$;
3. The *bootstrap inputs* H_0 is a set of predefined "seed" inputs before the search procedure starts.

In this tuple-based framework, a coverage-guided genetic fuzzing technique can be characterized by $\langle N_{\mu,\chi}, S_c, H_0 \rangle$, in which $\mu(H)$ mutates test inputs in H, $\chi(H')$ crosses over inputs in $H' \subseteq H$, and S_c assigns newly generated test inputs with priorities based on whether they reached previously non-covered code. On the other hand, a solver-based DSE technique can be described by $\langle N_\sigma, S, H_0 \rangle$, in which symbolic analysis σ is used to negate a branch to yield a new test input.

A Mini Survey of Existing Test Input Generation Techniques. This unified framework offers us a unique opportunity to study apparently different techniques under the same basis. Particularly this paper focuses on the *complementariness* and *common limitations* between fuzzing and DSE. This paper presents a survey of representative fuzzing and DSE techniques as well as their hybrids.

An empirical study. The qualitative analytical results are further validated by a quantitative empirical study of the testing performance of AFLfast and KLEE (as the representative techniques of fuzzing and DSE) over a set of GNU CoreUtils programs.

In the empirical study, both AFLfast and KLEE achieved a quite high level of statement coverage. The coverage of AFLfast (85.99% on average) is slightly higher than that of KLEE (81.18% on average).

We in-depth studied all 142 non-covered cases in which 56 cases are of insufficient modeling of system states, data types, or library/system APIs and 95 cases are due to the algorithmic limitation of the concerned search space definition or search strategy[1)]. Though we only quantitatively studied AFLfast and KLEE, the results are consistent with the analytical study and we thus believe that these results can be applied to a broader range of fuzzing and DSE based techniques.

Finally, we summarized our results as a series of findings for future researches on automated test input generation: (1) Different search strategies, particularly fuzzing and DSE, not only *complement* each other but also have *common limitations*. (2) There are *challenges* in exploiting the complementariness over different search strategies. (3) Simple code coverage may *not* be the proper criterion in guiding the search towards deeper states. (4) *Program transformation* fundamentally changes the search space and maybe a promising research direction. (5) There may be a *grand unified framework* to fully leverage the power of existing techniques on test input generation.

The rest of the paper is organized as follows. Section 2 introduces necessary background on the test input generation problem. Section 3 elaborates on our $\langle N, S, H_0 \rangle$ framework for describing an automated test input generation technique, followed by a mini survey of existing pure fuzzing/DSE techniques in Section 4. Section 5 presents the empirical study of applying AFLfast and KLEE to a set of GNU CoreUtils programs. Hybrid techniques are surveyed in Section 6. Sections 7 and 8 discuss the findings and related work, and finally Section 9 concludes this paper.

2 Background

2.1 Test Input Generation Problem

Given a program P, let T be the set of P's all possible test inputs and O be a test oracle[2)] where $O(t) \in \{\top, \bot\}$ (*true* or *false*) for $t \in T$. It is desirable to show that P is *correct* ($P \vdash O$), i.e., $\forall t \in T.\, O(t) = \top$. However, for real-world programs, it is neither practical to directly verify P (i.e., statically proving that $P \vdash O$) nor possible to exhaustively enumerate and validate $O(t)$ for all $t \in T$.

Therefore, *testing* becomes the only feasible way to find some clues on whether $P \vdash O$. In testing, O is validated against a small set of test inputs $T' \subseteq T$. Either there is an $t' \in T'$ and $O(t') = \bot$ indicating that $P \nvdash O$, or $\forall t' \in T'.\, O(t') = \top$ gives some confidence on $P \vdash O$. Testing never proves $P \vdash O$. Therefore, it is crucial to *thoroughly* test P using a *representative* set of test inputs T' to gain confidence on $P \vdash O$.

Automatically generating high-quality inputs for a thorough testing, i.e., the *test input generation* problem, is one of the Holy Grails in software engineering. Unfortunately, how to quantify the representativeness highly depends on the specification of P, and the existence of a general criterion is still in debate [14]. Practically, test thoroughness is approx-

[1)] The sum is greater than 142 because a statement covered by neither AFLfast nor KLEE may be of different causes.

[2)] Test oracle reflects the correctness of a program, e.g., P never crashes, P does not leak any memory, or P is always responding to its received requests.

imately quantified by the *test coverage* of T' [14, 15]. The rationale of test coverage is that if T' fails to cover something (a statement, a branch, of a define-use pair) in P, it has no chance to detect a bug in that program construct. Therefore, achieving a high test coverage is believed to be the least baseline of a thorough testing.

Therefore, a milestone of automated test input generation is generating test inputs to achieve a high test coverage under a particular test coverage criterion. This paper discusses two categories of mainstream approaches to automated test input generation: fuzzing and DSE.

2.2 Fuzzing and DSE

2.2.1 Fuzzing

Fuzzing generates a large amount of test inputs using random or heuristic rules to trigger unintended program behaviors (violating the test oracle O). This seemingly straightforward idea has been extensively studied and enhanced, and among which the *coverage-directed genetic* approach [9, 16–18] is one of the most successful techniques.

Such an approach keeps a runtime pool of inputs (and their coverage profiles) and uses heuristic rules to guide the generation of new inputs that may have a chance to reach non-covered code regions. Inputs in the pool are evaluated under *fitness functions*, which reflect to what extent an input is promising in the testing procedure[3)]. This paper focuses on this category of techniques because they are the most extensively studied, achieved promising experimental results, and are adopted by the industry (e.g., AFL [8] and libFuzzer [19]).

2.2.2 DSE

Dynamic symbolic execution (DSE) [4, 20] is a derivative of symbolic execution [21, 22], in which a program's all paths are symbolically analyzed. Instead of traversing all program paths, DSE exploits a constraint solver to drive the program under test to a particular path and executes the program (either with concrete inputs or using an emulator) to obtain more test inputs.

In theory, DSE can verify a program against a test oracle given infinite amount of time. In practice, heuristic rules have to be used for quickly identifying potentially useful program paths for exploring non-covered code.

[3)] A straightforward idea is that the *coverage guided* approach assigns in which assigns test inputs reaching previously non-covered code regions a higher fitness value.

3 Test Input Generation as a Search Problem

3.1 The Framework

Automated test input generation is challenging because the structure of T is extremely difficult to characterize, and obtaining test inputs to satisfy a certain property (e.g., following a particular program path, reaching a particular program point, or triggering a test oracle violation) is undecidable. Therefore, an automated test input generation technique could only afford generating new test inputs on the basis of a set of tested inputs (denoted by H) within limited computational resource budgets, which can be regarded as a *search algorithm* for sampling test inputs in T.

The most successful approaches, fuzzing and DSE, are both search algorithms but adopt quite different *search strategies*. Our later empirical study (in Section 5) supports the common belief that fuzzing and DSE are considerably different: there are substantial codes which one approach can easily cover but the other cannot.

To further understand *why* a piece of code can/cannot be covered by a particular technique, we present a framework to analytically study the characteristics, strength, and weakness of existing techniques (fuzzing, DSE, or even their combinations) under the same perspective. In this framework, the search strategy of each technique is described by $\langle N, S, H_0 \rangle$, (this is a generalization of the existing survey [23]) where:

1. the *neighborhood definition* N defines a set of test inputs that may be potentially beneficial for exploring more code on the basis of historically explored inputs H,
2. the *neighborhood selection* strategy S prioritizes (and prunes) test inputs in $N(H)$ to obtain newly generated inputs and drive the search procedure, and
3. a set of bootstrap test inputs H_0.

Under this framework, an automated test input generation technique is an iterative procedure, in which each iteration updates H by $S(N(H))$ starting from H_0, i.e.,

$$H_{i+1} \leftarrow H_i \cup S(N(H_i)).$$

When the testing procedure terminates, H_n in the last iteration contains all generated test inputs. This framework strives to separate the *mechanism* for conducting the search (by the neighborhood definition strategy N) from the actual *policy* of test input selection (by the neighborhood selection strategy S) [24]. As a trivial example, in the random testing described

```
1  void f(int x, int y) {
2    if (x == y) {
3      if (x + y == 0x4758216) {
4        // difficult to reach
5      }
6    }
7  }
```

Fig. 1: A code region difficult to reach by fuzzing.

```
1  int sum = 0, len = strlen(str);
2  for (int i = 0; i < len; i++) {
3    if (str[i] == 'x') sum++;
4  }
5  if (len == 100 && sum == 75) {
6    // difficult to reach
7  }
```

Fig. 2: A code region difficult to reach by both fuzzing and DSE. This example is from [32].

by $\langle N_r, S_r, H_0 \rangle$, N_r always returns a random test input regardless of H, and $S_r(N_r(H)) = N_r(H)$.

To the best of our knowledge, this framework could cover all our known existing techniques [9, 16–18, 25–39] and allows one to qualitatively study their characteristics in depth (Section 4) and quantitatively evaluate them using real-world programs (Section 5).

3.2 Basic Fuzzing and DSE in the Framework

The neighborhood definition N determines the structure of the search space, and the neighborhood selection S reflects the policy in the search algorithm. In the ideal case, we hope that any piece of code is easily reachable in the search space of N, and the search can be effectively guided by S.

Fuzzing defines the neighbours of H by applying *lexical changes* to given test inputs, which is the most straightforward approach to defining a search space. In particular, $N_{\mu,\chi}(H)$ consists of test inputs generated by mutation ($\mu(H) \subseteq T$ applies some random changes to some $t \in H$) or cross-over ($\chi(H') \subseteq T$ combines test inputs in $H' \subseteq H$, e.g., by concatenating alternative string fragments in H', usually for $|H'| = 2$).

Therefore, it is easy to generate a substantial amount of offspring given H and filter them with a given *fitness* threshold. $S(N(H))$ only contains those test inputs that have a relatively high fitness value, e.g., test inputs reaching a previously non-covered code region. This is essentially the genetic algorithm served as the basis of many modern fuzzers [8, 9, 17, 18, 18, 19, 29, 37, 39].

However, fuzzers usually cannot reach code regions where there lacks explicit feedback on the "distance" between a test input and the testing goal (e.g., reaching a piece of code), particularly when input variables are internally correlated. Such an example is shown in Fig. 1, in which covering Line 4 requires both $x = y$ and $x + y$ equal a particular value. Even though cross-over may effectively generate test inputs satisfying $x = y$, $\langle N_{\mu,\chi}, S \rangle$ lacks a mechanism to guide a fuzzer to set $x = y =$ `0x23ac10b`.

Dynamic symbolic execution (DSE), on the other hand, explores execution *paths* of a program P by iteratively exercising branch negations by a symbolic analysis engine σ provided with N_σ. Let $\varphi(t) = \{b_1, b_2, \ldots, b_m\}$ in which $b_i \in \{\top, \bot\}$ $(1 \leq i \leq m)$ denotes the i-th branch is taken/non-taken in the trace of executing program P with t. The symbolic analysis engine σ can *negate* the k-th branch of $\varphi(t)$, i.e., $\sigma(t, k)$ returns a test input t' whose $\varphi(t') = \{b_1, b_2, \ldots, b_{k-1}, \neg b_k, \ldots\}$, or reports that such an input does not exist. Therefore, test inputs in $N_\sigma(H)$ will follow a strictly different execution path, and DSE in theory could enumerate all possible execution paths of P without any duplication with earlier explored execution paths.

Symbolic analysis is a highly non-trivial technique implemented by replacing concrete values in t by symbolic variables and collecting each branch b_i's branch condition Φ_i (Φ_i is the constraint of i-th branch to be evaluated to b_i). $\sigma(t, k)$ is obtained by solving the path constraint

$$\Phi = \Phi_1 \wedge \Phi_2 \ldots \wedge \Phi_{k-1} \wedge \neg \Phi_k,$$

yielding t' to follow the designated execution path.

DSE is effective in finding test inputs to follow a particular path, e.g., the case in Fig. 1, however, real-world programs may have exponentially many execution paths and only a few of them may be of particular interest. An example is shown in Fig. 2, in which reaching Line 6 requires exactly 75 times of `str[i] == 'x'` and 25 times of `str[i] != x`. Negating the `sum == 75` branch is always infeasible because `sum` is control-dependent on previous branches.

4 Pure Fuzzing and DSE Techniques: A Mini Survey

The new perspective of considering an automated test input generation technique as $\langle N, S, H_0 \rangle$ enables us to study existing test input generation techniques inside a single, unified framework. Table 1 lists 17 surveyed pure-fuzzing and pure-DSE techniques [4]. Each technique in the table can be re-

[4] Automated test input generation is an active research field. This paper is not intended to exhaustively survey existing work, rather,

Table 1: Summary of typical existing automated test input generation techniques. The particularly innovative parts are denoted by a box.

Technique	Year	Summary	Distinctive Feature
SMART [26]	2007	$\langle N_{\boxed{\sigma}}, S, H_0 \rangle$	Accumulating symbolic function summaries (e.g., symbolic representation of $f(x)$ under a certain constraint $\psi(x)$) along with the dynamic symbolic execution of P, and re-using such summaries to accelerate future symbolic analyses (e.g., when $f(y)$ is evaluated with $\psi(y)$).
CREST [27]	2008	$\langle \boxed{N_\sigma}, \boxed{S}, H_0 \rangle$	Prioritizing static paths in the control flow graph of P according to their "distances" to currently covered branches and using such paths to guide the branch negation.
RWset [28]	2008	$\langle N_\sigma, \boxed{S}, H_0 \rangle$	Pruning a test input if it is redundant in terms of the read-write set (RWset) analysis, which exploits the observation that two program states at the same program location are equivalent if they will read identical variable values.
BuzzFuzz [29]	2009	$\langle \boxed{N_{\mu,\chi}}, S, H_0 \rangle$	Using the dynamic taint analysis to obtain possible "interesting" values of input bytes (e.g., a byte is compared against a magic number) to guide the input mutation strategy μ.
Fitnex [30]	2009	$\langle N_\sigma, \boxed{S}, H_0 \rangle$	Defining a fitness value in S based not only on coarse-grained coverage statistics but also fine-grained expression/function evaluation information to reflect how much a branch condition is satisfied.
BFAFI [39]	2011	$\langle \boxed{N_{\mu,\chi}}, S, H_0 \rangle$	Learning a model of the input file by profiling program executions, and generating only test inputs consistent with the model in $N_{\mu,\chi}$.
Sub-path DSE [31]	2013	$\langle N_\sigma, \boxed{S}, H_0 \rangle$	Guiding the neighborhood selection by an extended concept of branch coverage: k-subpath coverage. Branch negation is prioritized by the infrequency of existing k-subpaths.
MULTISE [33]	2015	$\langle \boxed{N_\sigma}, \boxed{S}, H_0 \rangle$	Defining the search space in terms of program *states* including program location (e.g., $pc \mapsto 10, x \mapsto \{(cond, v_1), (\neg cond, v_2)\}$) instead of paths.
GUIDESE [34]	2015	$\langle N_\sigma, \boxed{S}, H_0 \rangle$	Using user-provided annotations to model program states for state merging and pruning, and guiding the exploration of designated paths.
DASE [35]	2015	$\langle \boxed{N_\sigma}, S, \boxed{H_0} \rangle$	Using input constraints obtained from textural documents to obtain meaningful seeds and prune invalid test inputs in $N(H)$.
DCA [36]	2015	$\langle N_{\boxed{\sigma}}, S, H_0 \rangle$	(Don't Care Analysis) Identifying redundant assignment statements that do not affect code coverage to accelerate constraint solving in the symbolic analysis.
MoWF [37]	2016	$\langle \boxed{N_{\mu,\chi}}, S, \boxed{H_0} \rangle$	Using a user-provide test input model to constrain the inputs in H_0 and generating only model-satisfying inputs in μ and χ.
AFLfast [9]	2016	$\langle N_{\mu,\chi}, \boxed{S}, H_0 \rangle$	Prioritizing mutation/crossing of test inputs in H by their path infrequencies, i.e., $\Pr[t \in H \text{ is chosen}] \propto 2^{-\text{freq}(\varphi(t))}$, where $\text{freq}(\varphi(t))$ denotes the frequency of $\varphi(t)$ occurring in H.
Steelix [18]	2017	$\langle N_{\mu,\chi}, \boxed{S}, H_0 \rangle$	Defining a fitness score in S based not only on coarse-grained coverage statistics but also on fine-grained expression/function evaluation information (e.g., the number of iterations conducted in `strncmp`).
VUzzer [17]	2017	$\langle \boxed{N_{\mu,\chi}}, \boxed{S}, \boxed{H_0} \rangle$	Adopting multiple optimizations, e.g., identifying error-handling basic blocks and reducing their fitness values to increase the chance of generating valid inputs, and using interesting values obtained in the taint analysis.
LEO [40]	2018	$\langle \boxed{N_\sigma}, S, H_0 \rangle$	Using heuristics rules to selectively optimize the source code, yielding an alternative search space which may be easier to explore.
ParaDySE [41]	2018	$\langle N_\sigma, \boxed{S}, H_0 \rangle$	Before conducting formal symbolic analysis, try to learn the optimal parameterized search heuristic based on specified branch features for the current subject program.

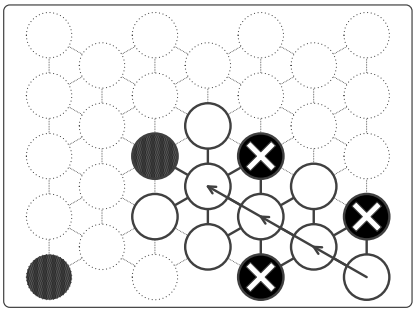

Fig. 3: Illustration of search space pruning. Redundant or invalid test inputs in N (marked by $\times$ in the figure) are identified *before* the neighborhood selection (S) and pruned.

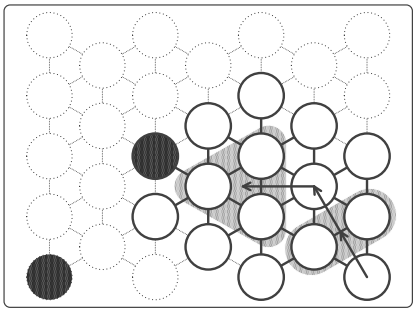

Fig. 4: Illustration of state merging. Test inputs leading to the same program state are merged (denoted by a shadowed area in the figure) in generating subsequent test inputs.

garded as some *optimizations* to N, S, or H_0 to address some particular limitations of a previous technique. This section presents the survey and analytical study of these techniques.

4.1 Optimizing the Search: The Mini Survey

In the search space of N_σ, a new test input can be obtained by selecting and negating a branch in the execution trace of a test input in H. As the execution trace can be very long, the effectiveness of DSE depends highly on to what extent S can select a potentially beneficial branch to negate. It is natural to apply the idea of "coverage guided" in fuzzing to prioritize candidate branches to negate: CREST [27] and Fitnex [30] calculated the logical distance towards a piece of non-covered code, and sub-path DSE extended the idea of branch coverage to k-subpath coverage.

Observing that the symbolic analysis engine σ is the most costly component in DSE, SMART [26] conducted compositional DSE to summarize constraints along an execution path, and accelerated the execution speed by re-using the summary. DCA [36] identified the "don't-care" variables, which do not affect the execution path, and ignored redundant assignment statements arising from the don't-care variables.

On the other hand, obtaining a new test input by negating a branch (simultaneously keeping all previous branches unchanged) is not always feasible, as shown in Fig. 2, and exhaustively enumerating exponential many possible paths is also infeasible. Such an intrinsic limitation of branch-negation based symbolic analyses yields *merging* of paths, i.e., focusing on the program *state* at a particular program point. RWset [28] viewed two program states at the same program location as *equivalent* if they will read identical variable values, and prunes redundant paths (Fig. 3). GUIDESE [34] exploited user-provided annotations to guide the state merging and path pruning. DASE [35] obtained constraints from textural documents, based on which to construct valid inputs. MULTISE [33] conducted state merging to guide the exploration towards designated paths. If two paths reached at a particular program counter and have a similar variable state, they may be merged (considered equivalent) in conducting the future search, as illustrated in Fig. 4.

The search space $N_{\mu_\mathcal{X}}$ of fuzzing, on the other hand, is clearly defined and easy to compute. Therefore, the optimizations of fuzzing focus mainly on the neighborhood selection strategy S. A representative example is AFLfast [9], which gave a high priority to the inputs with infrequent paths on breeding next generations.

To exploit fine-grained information in evaluating the fitness value of an input, BuzzFuzz [29] and VUzzer [17] exploited the taint analysis to guide the mutation. Based on the taint analysis, interesting values are extracted from the source code and used to guide the mutation on the interesting bytes in the input.

An orthogonal optimization is to expose fine-grained program logic for both fuzzing or DSE to easier reach deep code blocks, either by refining the coverage criteria to reflect the internal logic in a function (e.g., a `strcmp`) as adopted in Steelix [18], or by performing designated program transformations as adopted in LEO [40]. Similar difficulties maybe alleviated by leveraging an input model (Fig. 3). MoWF [37] used a user-provided model to partially constrain the inputs to generate only model-satisfying inputs. BFAFI [39] automatically learned a model of the input file by profiling program executions.

4.2 Discussions

Examining the existing test input generation techniques in the chronological order reveals that fuzzing and DSE tech-

niques are undergoing extensive improvements, and are becoming becoming *increasingly sophisticated*. As opposed to early research mainly focused on improving the search strategy (e.g., exploiting fine-grained profiling information [30]) or pruning the search space (e.g. by tainting bytes [29] or model [39]), many recent techniques consider the optimization of $\langle N, S, H_0 \rangle$ as an integral part.

On the other hand, some latest research also shows that there is still much to explore in tuning the search strategy S [18], and even simple heuristics may significantly improve the performance of an automated test input generation tool [9]. Therefore, it is crucial for us to investigate the intrinsic limitation of fuzzing and DSE on real programs, which is conducted as follows.

5 Understanding the Limitation of Fuzzing and DSE: An Empirical Study

We are particularly interested in the characteristics of covered and non-covered code of fuzzing and DSE techniques in testing real-world programs. This section presents our empirical study of such code by applying AFLfast and KLEE to a set of GNU CoreUtils programs.

5.1 Study Design

Q2 in Section 1 can be further expanded into the following research questions:

- What code is covered or not covered by fuzzing and DSE, respectively?
- Is there code difficult to cover by a technique that can be easily covered by another?
- Are there characteristics of code that cannot be covered by existing techniques?

We chose AFLfast and KLEE as the representative techniques of fuzzing and DSE, respectively. We applied AFLfast and KLEE to 19 GNU CoreUtils programs[5], which do not accept files as their arguments, and studied their statement coverage performance (collected by `gcov`). We manually analyzed each piece of covered and non-covered[6] code and summarized their characteristics.

[5] GNU CoreUtils is one of the most mature command-line utility collections, which well represents the scenario of testing a functional part in a practical system. The subject programs are all from the latest version (8.27).

[6] We only study blocks of code that are *directly* non-covered. If a piece of non-covered code is control dependent on another piece of non-covered code (e.g., its upstream branch is not covered), we exclude it from our study because it may *not* be difficult to cover.

Table 2: Statement Coverage Comparison of AFLfast and KLEE. A starred number denotes the highest coverage. A number in the bracket shows its gap from the highest coverage.

	Statement Coverage (%)		
Program	**AFLfast (`a`)**	**AFLfast (`-help`)**	**KLEE**
`basename`	76.9 (-23.1)	⋆100	⋆100
`date`	77.2 (-11.7)	⋆88.9	88.0 (-0.9)
`dirname`	72.7 (-27.3)	⋆100	⋆100
`echo`	81.9 (-14.9)	⋆96.8	78.9 (-17.9)
`expr`	68.1 (-16.6)	76.8 (-7.8)	⋆84.7
`factor`	91.7 (-1.3)	⋆93.0	33.7 (-59.4)
`hostid`	81.8 (-18.2)	⋆100	⋆100
`logname`	70.8 (-20.8)	⋆91.7	⋆91.7
`nproc`	68.8 (-28.6)	95.0 (-2.3)	⋆97.3
`numfmt`	51.5 (-14.5)	63.0 (-2.9)	⋆65.9
`pathchk`	78.2 (-2.7)	⋆80.9	72.0 (-8.9)
`printenv`	97.4 (-2.6)	⋆100	⋆100
`printf`	83.3 (-8.5)	91.6 (-0.2)	⋆91.8
`pwd`	24.2 (-9.4)	32.0 (-1.5)	⋆33.6
`seq`	72.6 (-13.2)	⋆85.8	78.2 (-7.6)
`tr`	51.6 (-15.4)	⋆67.0	60.5 (-6.5)
`uname`	82.1 (-9.3)	⋆91.4	90.9 (-0.5)
`who`	82.2 (-1.3)	⋆83.5	79.2 (-4.3)
`whoami`	77.8 (-18.5)	⋆96.3	⋆96.3
Average	**73.20**	**⋆85.99**	**81.18**
Median	**77.23**	**⋆91.62**	**87.95**

We selected only CoreUtils programs whose inputs are purely from command-line arguments (i.e, do not relate to any file content) to ensure that there is a *single source of inputs* for a fair comparison[7]. This is because AFLfast made such an assumption and may fall short when crossing over multiple sources of inputs. Furthermore, a single source of input does not imply that the subject programs are easy to thoroughly test: arguments may work in combinations or indicate a complex expression.

For each technique, we suppose that it is deployed in a fully automated testing scenario in which the tester provides *minimal* human domain knowledge of a program under test. For AFLfast, we gave it either a trivial but valid argument "`-help`" or a probably invalid argument "`a`" as the bootstrap seed input. For KLEE, we simply ran it without any auxiliary input (only with its built-in system library model). Each technique is ran under a one-hour time limit on a commodity PC with a dual-core Intel i5 processor and 4GB RAM to represent a typical budget-limited testing scenario.

5.2 Covered Statements

The overall coverage results are shown in Table 2. It is clear that both fuzzing and DSE achieved a quite high code

[7] `factor` and `tr` are included because they receive relatively simple command line arguments which can be fixed in testing.

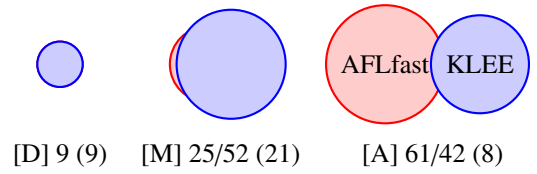

Fig. 5: Statistics of the non-covered cases. For each category, *a* / *b* denotes the number of non-covered cases of AFLfast (red) / KLEE (blue), respectively. The number in a bracket denotes the number of shared non-covered cases.

coverage, and the fuzzing-based AFLfast even outperformed KLEE, which is particularly tuned for CoreUtils with many libraries modeled.

The results also show that fuzzing is quite sensitive to its bootstrap seed inputs (i.e., the initial H). Even a trivial but valid input (`-help`) led to a significant statement coverage advantage (12.8% on average) over a probably invalid input (`a`). Therefore, incorporating the knowledge contained in regression test inputs to facilitate an even more effective automated testing may worth a deeper study. In the rest parts of the empirical study, we focus only on the coverage performance of AFLfast under an effective seed input (`-help`).

5.3 Non-covered Statements and Their Characteristics

5.3.1 Overall Results

We manually extracted all 142 non-covered cases by filtering out indirectly non-covered statements (a non-covered code block that is control-dependent on another non-covered statement), and removed five non-covered cases, which are due to `gcov` collecting incomplete coverage statistics on a program's timeout.

We summarize the causes of these non-covered cases into three categories:

- [D] Dead code that denotes a situation not expected to happen, e.g., an algorithm proven to always success fails to find a solution.
- [M] Insufficient modeling of system states, data types, library or system call APIs, whose analyses are required to define a complete search space or effectively guide the search procedure.
- [A] Algorithmic limitation of the search space, strategy, or bootstrap input such that reaching the concerned code is extremely costly. In theory, such cases may be covered given an infinite amount of time. However, they have little chance to be covered in a resource-limited practical setting.

The overall statistics of all cases are shown in Fig. 5, and we summarize two dead code cases (D) and twenty insufficient modeling (M) or algorithmic limitation (A) cases in Table 3. The latter is elaborated on as follows.

5.3.2 Insufficient Modeling

In the following case of `pwd`:314, the program execution depends on an environment variable `PWD`. Both the AFLfast and KLEE implementations did not explicitly model such states (and their correlations to program executions). In other words, the system states that will affect program executions are *not* defined in the search space of N. Nevertheless, what system states should or should not be modeled is always a challenge for implementing an automated test input generation tool.

`pwd:314`

```
1  static char *logical_getcwd (void) {
2    ...
3    char *wd = getenv ("PWD");
4    char *p;
5    p = wd;
6    while ((p = strstr (p, "/."))) {
7      // non-covered code
8    }
9  }
```

Another similar case from `numfmt`:1457 requires an empty radix character, which is related to the system's locale setting. If such environment-dependent code may lead to a bug, it could only be manifested in production systems in rare scenarios (e.g., the system administrator or user sets an invalid locale), and may be extremely difficult to reproduce and localize.

`numfmt:1457`

```
1  decimal_point = nl_langinfo(RADIXCHAR);
2  if (decimal_point == NULL ||
3     strlen(decimal_point) == 0) {
4     // dependent on locale settings
5  }
```

In some cases, the program may use data types (or operations) that are not supported by a symbolic analysis engine. For example, `expr`:103 uses 64-bit integer comparisons, which are not supported by the DSE engine of KLEE, even if the path constraint is quite trivial from a human's perspective. This case is also not covered by AFLfast due to its algorithmic limitation: a coverage directed search cannot effectively guide the setting of values of `a` and `b` towards satisfying the branch condition. However, fuzzing generally suffers less from such data modeling issues, e.g., `seq`:153 and `factor`:789 are both covered by AFLfast.

Table 3: A selection of typical non-covered cases. CF/CD denotes the code is covered by AFLfast/KLEE, respectively.

#	Location	CF / CD	Non-CF Cause	Non-CD Cause
1	`expr`:448	×/×	[D] the `default` branch of a switch-case is unreachable	
2	`factor`:1376	×/×	[D] asserts that the algorithm will not fail	
3	`seq`:153	✓/×		[M] requires a non-number argument and `isnan` is not properly modeled
4	`factor`:789	✓/×		[M] involves 64-bit integer arithmetics that is not supported by σ
5	`factor`:2408	×/✓	[M] requires that standard input should be a terminal, i.e., `isatty` returns 1	
6	`numfmt`:1457	×/×	[M] requires the current locale to have an empty radix character	
7	`pwd`:314	×/×	[M] requires an non-absolute `PWD` environment variable	
8	`seq`:308	×/×	[M] requires `printf` to fail in writing `stdout`	
9	`expr`:573	×/×	[M] requires `re_match` to return a successful match	
10	`echo`:137	×/✓	[A] requires a standalone `-version` argument without any other option	
11	`expr`:392	×/✓	[A] requires an arithmetic expression of a particular structure with parentheses, e.g., `x + ( y + z )`	
12	`numfmt`:1504	×/✓	[A] requires `-padding` with a negative number	
13	`pathchk`:271	×/✓	[A] requires `-p` or `-P` followed by an empty string	
14	`printf`:282	×/✓	[A] requires a unicode in the escape format, e.g., `\u1234`	
15	`expr`:651	×/✓	[A] requires the first argument equal to `length` followed by a string	
16	`numfmt`:1557	×/✓	[A] requires `-format=` present in the arguments	
17	`printf`:433	✓/×		[A] requires a particular format string
18	`numfmt`:494	✓/×		[A] requires a large number more than 18 digits, which is parsed by a loop
19	`pathchk`:405	✓/×		[A] requires a long file name exceeding `name_max`
20	`expr`:103	×/×	[A] requires a `INT64_MAX/-1` number	[M] KLEE cannot support 64-bit integer arithmetics
21	`factor`:767	×/×	[A] requires an input to be a multiple of 2^{64}	[M] KLEE cannot support 64-bit integer arithmetics
22	`printf`:458	×/×	[A] requires a format string of `%.*s`, which is parsed in a loop	

`expr:103`

```
1  void mpz_tdiv_q (mpz_t r, mpz_t a0, mpz_t b0) {
2    intmax_t a = a0[0];
3    intmax_t b = b0[0];
4    if (a < - INTMAX_MAX && b == -1)
5      // failed to solve
6      integer_overflow ('/');
7    ...
8  }
```

In the 25/52 non-covered cases of AFLfast/KLEE due to such insufficient modeling, there are 21 common cases. Failing to model system states is one of the major limitations of applying automated test input generation techniques in system testing of real-world programs, as these programs may extensively interact with their external environments: the running environment, the file system, or even a server on the Web.

5.3.3 Algorithmic Limitations

There can be a piece of code that requires a particular program state to cover at a branch. However, existing heuristics may not be able to effectively guide the discovery of such a state. `printf`:458 is one of such cases (covered by neither technique).

In this example, the constraint ¬`have_field_width` and `have_precision` must be simultaneously satisfied, i.e., requiring the system state to satisfy a particular combination. Unfortunately, the coverage-directed fitness functions in both AFLfast and KLEE are not aware of such a state, leading to non-covered code. For KLEE, the symbolic analysis cannot negate the branch condition while not affecting the previous control flow, which is similar to the case in Fig. 2.

`printf:458`

```
1  for (f = format; *f; ++f) {
2    switch (*f) {
3      case '\%':
4      ...
5      print_direc(..., have_field_width,
6                       have_precision, ...);
7      ...
8    }
9  }
10 static void print_direc(...) {
11   .....
12   switch (conversion) {
13   case 's':
14     if (!have_field_width) {
15       if (have_precision) {
16         // a specific format string, e.g., "%.*s"
17       }
18     }
19   }
```

```
20  }
```

On the other hand, the coverage-directed heuristics and mutation/cross-over between inputs did make fuzzing cover more code by "luck", as in the example of `printf`:433. KLEE did not cover this case for the same reason of not covering `printf`:458.

`printf:433`

```
1  case 'G':
2    if (!have_field_width) {
3      if (!have_precision)
4        xprintf (p, arg);
5      else
6        // AFLfast covered this line
7    }     else {
8      ...
9    }
10   break;
```

The local search nature of fuzzing techniques also comes with limitations. `pathchk`:271 reveals such an algorithmic limitation. When several test inputs are *correlated*, e.g., requiring `-p` or `-P` followed by an empty string in this case, it would require multiple steps of mutation (μ) or cross-over (χ) to generate even a valid test input. In case that the search strategy fails to identify the correlations and test inputs in an intermediate step may have a particularly low fitness value, generating such test inputs would be extremely challenging[8]. In contrast, such correlations can be easily handled by symbolic analysis, and thus KLEE successfully covered this case.

`pathchk:271`

```
1  bool validate_file_name(char *file,
2    bool check_basic_portability,
3    bool check_extra_portability) {
4    ...
5    if ((check_basic_portability ||
6      check_extra_portability) &&
7      filelen == 0) {
8        error(0, 0, _("empty␣file␣name"));
9        return false;
10   }
11   ...
12 }
```

In the 61/42 non-covered cases of AFLfast/KLEE due to algorithmic limitations, only 8 cases are overlapping. In other words, the search spaces and strategies of $\langle N_{\mu,\chi}, S, H_0\rangle$ and $\langle N_\sigma, S, H_0\rangle$ behave quite differently in covering those "difficult to reach" code. Therefore, purely enhancing the automated test input generation techniques with more precise modeling or more powerful symbolic analysis engine could be still insufficient for a thorough testing.

[8] In this case, fuzzing separately generated `-p`, `-P`, and an empty string, but failed to generate their combinations.

5.4 Discussions

We believe that the empirical study results, even though conducted on the baseline off-the-shelf versions of AFLfast and KLEE without state-of-the-art optimizations in the empirical study, revealed some key limitations of fuzzing and DSE techniques.

First, the insufficient modeling issue will exist as long as one cannot provide precise model of all libraries and system calls used in the program under test. This limitation is largely application- and language-dependent (e.g., `libc` is relatively easier to model compared with `Javascript` standard libraries), and therefore is not the current major focus of the research community.

For the algorithmic limitations issue, we found that there are deeply correlated input bytes which are extremely difficult to cover (e.g., `printf`:458) and unfortunately this algorithmic limitation seems not to be easily resolved even with state-of-the-art optimization techniques.

However, the experimental results also revealed that fuzzing and DSE indeed "favor" covering code of distinct characteristics and complement each other, and there is no surprise that considering multiple search spaces (e.g., $N_{\mu,\chi} \cup N_\sigma$) is a promising direction towards more effective test input generation.

6 Hybrid Test Input Generation: A New Research Frontier

The idea of hybrid test input generation dates back to early research of dynamic symbolic execution [25]: and it is straightforward to combine multiple test input generation techniques by letting S to freely choose test inputs in multiple search spaces: N_r in random testing, $N_{\mu,\chi}$ in fuzzing or N_σ in DSE. This simple yet powerful idea has recently been extensively developed to further improve the effectiveness of automated test input generation. We surveyed typical hybrid techniques as listed in Table 4.

6.1 The Hybrid Approach: A Mini Survey

The empirical study results in Section 5 show that fuzzing and DSE have their distinct strengths and limitations in terms of covering code lines, and a hybrid approach tries to use multiple search strategies to *compensate* each other for more effective test input generation.

The local-search nature of fuzzing makes it difficult to generate valid inputs that satisfy certain semantics constraints

Table 4: Summary of hybrid test input generation techniques.

Technique	Year	Summary	Distinctive Feature
Hybrid-Concolic [25]	2005	$\langle N_r \cup N_\sigma, S, H_0 \rangle$	Invoking a DSE engine to negate a difficult-to-reach branch if there has been a long time of random testing without any coverage gain.
TaintScope [42]	2011	$\langle N_{\mu_\chi} \cup N_\sigma, S, H_0 \rangle$	Identifying checksum fields and fill them with symbolic execution techniques, and applying fine-grained dynamic tainting to favor hot bytes related to security-sensitive operations.
VeriTesting [32]	2014	$\langle N_\sigma \cup N_{\text{SE}}, S, H_0 \rangle$	Conducting symbolic summarizations of functions and reducing the number of branches in an execution trace, yielding a more effective branch negation.
IC-Cut [38]	2015	$\langle N_\phi \cup N_\sigma, S, H_0 \rangle$	Conducting symbolic analysis to one function at a time with a single calling context on the basis of dynamically collected function summaries (ϕ_f of function f).
SYMFUZZ [43]	2015	$\langle N_{\mu_\chi}, S, H_0 \rangle$	Leveraging symbolic analysis to detect dependencies among the input bit positions, and dynamically tunes the mutation ratio with it in order to maximize the number of bugs found for black-box mutational-based fuzzing given the subject program and seed inputs.
Driller [16]	2016	$\langle N_{\mu_\chi} \cup N_\sigma, S, H_0 \rangle$	Invoking a DSE engine to negate a difficult-to-reach branch if there has been a long time of fuzzing without any coverage gain.
Mayhem [44]	2016	$\langle N_{\mu_\chi} \cup N_\sigma, S, H_0 \rangle$	Combining fuzzing and DSE, and invoking a DSE engine to bypass difficult-to-reach branch if there has been a long time of fuzzing without any coverage gain.
Basilisk-Framework [45]	2017	$\langle N_{\mu_\chi} \cup N_\sigma, S, H_0 \rangle$	Performing system-level fuzzing exploration and unit-level symbolic execution. Symbolic execution would be applied to parts of the system, which is hard-to-reach by the system-level fuzzing exploration.
Munch [46]	2018	$\langle N_{\mu_\chi} \cup N_\sigma, S, H_0 \rangle$ $\langle N_{\mu_\chi}, S, H_0 \rangle$	Interleaving fuzzing and DSE in two modes: FS hybrid mode for using directed DSE to reach non-covered functions by a fuzzer; and SF hybrid mode for using DSE to provide seeds for a fuzzer.
QYSM [47]	2018	$\langle N_{\mu_\chi} \cup N_\sigma, S, H_0 \rangle$	Accelerating the hybrid fuzzing-DSE by only performing symbolic execution on the symbolically tainted instructions, and only flips the constraints relevant to the target hard-to-satisfied branches by fuzzing exploration.
SAFL [48]	2018	$\langle N_{\mu_\chi}, S, H_0 \rangle$	Adopting symbolic analysis to generate high-quality seed inputs for the fuzzer, and prioritizing the fuzzing towards seeds which have exercised rarely executed branches.
T-Fuzz [49]	2018	$\langle N_{\mu_\chi}, S, H_0 \rangle$	Using program transformation to unsoundly remove hard-to-satisfy path constraints for fuzzer to explore designated parts of the state space, and use DSE to synthesize constraint-satisfying test inputs.

on the test input space. A widely discussed scenario is fuzzing a structural input with a checksum. Unless particularly treated [49], the checksum computation logic makes fuzzing almost always generating "trivial" inputs failing the checksum check.

Symbolic analysis is capable of synthesizing test inputs that satisfy such semantic constraints, but on the other hand is significantly slower in generating an input in N_σ compared with syntax mutation based fuzzing. Consequently, fuzzing can exercise much more test inputs given a same time budget.

Observing that *many code regions can be easily exercised by both techniques*, researchers are motivated to *interleave* fuzzing and DSE in various fashions.

The simplest form of interleaving fuzzing and DSE is to use one technique to provide seed inputs H_0 for another. For example, SYMFUZZ [43] conducted symbolic analysis before fuzzing to detect dependencies among the input bit positions, which are later used to tune the parameters in fuzzing (e.g., mutation ratio). SAFL [48] and Munch [46] worked in much similar way. SAFL invoked lightweight symbolic execution to generate qualified seeds, while Munch started fuzzing exploration for a limited time to amortize the cost of DSE in reaching hard-to-cover branches.

Distinct techniques can also be cooperatively *integrated* during the search procedure, i.e., by letting S to select test inputs from $N_{\mu_\chi} \cup N_\sigma$. In this mode, S may decide whenever one particular search strategy maybe more profitable than others: fuzzing would be preferred when it is quickly accumulating code coverage; however, when fuzzing fails in reaching particular code blocks for a long time, a heavyweight DSE may be helpful. This observation motivates a line of research work: Hybrid Concolic Testing [25], Basilisk-Framework [45], Driller [16], and Mayhem [44]. Both Driller and Mayhem took the heuristics approach of integrating DSE engine into fuzzer by invoking symbolic analysis when the coverage gain of the fuzzer becomes marginal.

The power of input constraint solving in symbolic analysis also facilitated integration of multiple techniques. Static

Table 5: Coverage Results of the AFLfast-KLEE Hybrid

	Statement Coverage (%)		
Program	**AFLfast**	**KLEE**	**Hybrid**
basename	⋆100.0	⋆100.0	⋆100.0
date	⋆88.9	88.0	79.8 (-9.1%)
dirname	⋆100.0	⋆100.0	⋆100.0
echo	96.8	78.9	⋆98.9 (+2.1%)
expr	76.8	84.7	⋆90.0 (+5.3%)
factor	⋆93.0	33.7	92.6 (-0.4%)
hostid	⋆100.0	⋆100.0	⋆100.0
logname	⋆91.7	⋆91.7	⋆91.7
nproc	95.0	97.3	⋆100.0 (+2.7%)
numfmt	63.0	65.9	⋆75.6 (+9.6%)
pathchk	80.9	72.0	⋆84.6 (+3.6%)
printenv	⋆100.0	⋆100.0	⋆100.0
printf	91.6	91.8	⋆93.7 (+1.9%)
pwd	32.0	⋆33.6	31.5 (-2.1%)
seq	85.8	78.2	⋆87.7 (+1.9%)
tr	67.0	60.5	⋆72.9 (+6.0%)
uname	91.4	90.9	⋆91.7 (+0.2%)
who	83.5	79.2	⋆84.4 (+0.9%)
whoami	⋆96.3	⋆96.3	⋆96.3
Average	**85.99**	**81.18**	**⋆87.96**
Median	**91.62**	**87.95**	**⋆91.67**

symbolic execution is a powerful tool to be integrated in DSE. Symbolic function summaries simplifies the path constraints to be solved in VeriTesting [32] while runtime information facilitates context-sensitive static symbolic analysis in the reverse direction in IC-Cut [38]. T-Fuzz [49], on the other hand, unsoundly reduced the program under test to a core logic (which is extensively fuzzed), and later used DSE to generalize the fuzzer-produced inputs for the core logic to the original program. In our framework, T-Fuzz can be treated as an extension to $N_{\mu,\chi} \cup N_\sigma$ where $N_{\mu,\chi}$ and N_σ are defined in the reduced and original versions of the same program, respectively.

6.2 Hybrid AFLfast and KLEE: An Empirical Study

We empirically validate the effectiveness of hybrid techniques by a study on a hybrid implementation similar to SAFL [48] and Munch [46] that use symbolic execution to provide seeds for a fuzzer. The hybrid interleaves AFLfast and KLEE (studied in Section 5) for a limited period of time to generate a more diverse set of bootstrap inputs H_0, and uses the lightweight fuzzing to perform further state space searching. H_0 is obtained by running each of AFLfast and KLEE for 10 minutes, and we retained the generated inputs of KLEE that cover AFLfast's non-covered code. We provide these inputs as the seeds for AFLfast (a typical $\langle N_{\mu,\chi}, S_c, H_0 \rangle$) for further 40 minutes of executions, to meet the same one-hour time limit and study the same statement coverage results, as shown in Table 5.

Overall, the AFLfast-KLEE hybrid outperforms both solely AFLfast or KLEE. The hybrid achieved the highest coverage for 16/19 (84%) programs, and has a coverage gain up to 9.6% compared with the best of AFLfast and KLEE. Some pieces of code were covered even if they involve quite complex input constraints, e.g., the following branch in `seq`:337.

`seq:337`

```
1  if (xstrtold (x_str + layout.prefix_len,
2    NULL, &x_val, c_strtold) && x_val == last) {
3      // covered only by the hybrid
4      ...
```

However, the hybrid is not capable of covering code beyond the capability of both techniques, e.g., requiring a particular system call to fail (`seq`:308) or involves in deep program logic (`printf`:458).

6.3 Discussions

$N_{\mu,\chi}$ and N_σ reflect two different perspectives on the same search space. Hard-to-reach test inputs in $N_{\mu,\chi}$ can be easily generated in N_σ, and vice versa. The empirical study validated that hybrid techniques would considerably improve the effectiveness of test input generation; however, hybrids are not the silver bullet to conquer those truly difficult cases (e.g., caused by insufficient modeling or when both algorithms fall short). The major challenge of adaptively integrating techniques is the increased difficulty in finding effective search strategy S.

Currently, integration of fuzzing and DSE is dominated by simple heuristic interleaving that worked well in practice, and it is still open whether there is an optimal strategy with theoretical guarantee (like that for DSE [50]) to maximize the profits of cooperating with both parties. There are a few pioneering work which may lead to further potential of research, and we discuss them as follows.

7 Findings and Implications

Finding 1. Different search strategies, particularly fuzzing and DSE, not only *complement* each other but also have *common limitations*.

Both the analytical study and empirical evaluation support the argument that the search spaces of fuzzing and DSE complement each other in finding useful test inputs in terms of covering more program states. Letting $N = N_{\mu,\chi} \cup N_\sigma$ may

reach program states that either technique alone cannot reach, and even simply interleaving two techniques can largely improve the test coverage.

On the other hand, both fuzzing and DSE face intrinsic challenges from modeling black-box functions and system states. There lacks an automated solution to the problem other than providing a manual stub or model. This limits the large-scale easy deployment of automated test input generation techniques. Attempts on exploiting external information [35] achieved encouraging results.

Finding 2. There are *challenges* in exploiting the complementariness over different search strategies.

In theory, combining techniques is merely proposing a neighborhood selection strategy S that adaptively invokes a technique that is expected to be the most profitable [16]. The actual challenge beneath the surface is what test input should be mutated, crossed over, or branch negated. Existing approaches made some preliminary attempts, but how to combine existing techniques for a balanced testing effectiveness and efficiency beyond simple heuristics is yet to be studied.

Essentially, this requires one to *predict* the outcome of performing a particular testing decision–either by formal methods (e.g., symbolic summary [32]) or statistics (e.g., machine learning [40]). We are expecting research work emerge towards this direction.

Finding 3. Simple code coverage may *not* be the proper criterion in guiding the search towards deeper states.

The empirical study exposed that there are deep program states dependent on complex input constraints. Currently, even state-of-the-art hybrid techniques are not expected to reach these code regions. This limitation may not easily be addressed because any search algorithm $\langle N, S, H_0 \rangle$ at each iteration could only examine a limited set of test inputs (and their execution traces) in H. As there may be little correlation between the testing "goal" $\Phi(t)$ (e.g., $\Phi(t) = \top$ for t reaching a particular line of code or triggering an oracle violation) and the current test inputs in H, any search strategy S would have difficulty in selecting effective test inputs in $N(H)$ towards the satisfaction of $\Phi(t)$.

To alleviate this issue, we believe that the test coverage should not be the single criterion of guidance–modeling of fine-grained program internal states is a promising direction [18]. A benchmark of parameterized "difficulty" (e.g., the depth of recursion, the number of correlated variables, etc.) may also be of particular interest to objectively evaluating a technique's strengths and weaknesses.

Finding 4. *Program transformation* fundamentally changes the search space and maybe a promising research direction.

This paper models the test input generation problem as exercising test inputs in program P's input space. Under this framework, $\langle N, S, H_0 \rangle$ suffices explaining the trade-offs in the search. Recent work suggests that performing transformation on P also significantly changes the search space's characteristics. Performing equivalent transformations to P has a similar effect of changing coverage criteria [40]. On the other hand, the case of T-Fuzz [49] showed that even aggressive (and unsound) *reduction* of P is capable of bypassing many sanity checks and lets the fuzzer focus on core parts of P's logic.

This is a relative new and emerging branch of automated test input generation. Particularly, the idea of generating test inputs for a portion of a program is much easier than performing fuzzing/DSE on the entire program. Program transformation techniques may be promising in decomposing a large testing problem into smaller sub-problems, and finally assembling the results back.

Finding 5. There may be a *grand unified framework* to fully leverage the power of existing techniques on test input generation.

Finally, the Holy Grail of automated test input generation lies in defining $\langle N, S, H_0 \rangle$ such that for a certain property $\Phi(t)$ concerned by the tester, there exists a sequence of adjacent test inputs connecting H_0 and test inputs satisfying $\Phi(t)$, and their corresponding fitness values are nearly monotone increasing. Theoretically, this is impossible for Turing-Complete programs. However, we have already witnessed the success of coverage-guided fuzzing or symbolic analysis in finding bugs and vulnerabilities. Hope that the perspectives in this paper may contribute to future development of effective test input generation techniques.

Reviewing all studied techniques, we found that they are largely *orthogonal* in improving automated test input generation in a particular facet. Therefore, it would be possible to define a search space unifying all existing ideas, e.g., $N = N_r \cup N_{\mu,\chi} \cup N^{\star}_{\mu,\chi} \cup N_{\sigma} \cup N_{\phi}$, adaptively switching between a number of existing search strategies S, and applying modeling or documentation for effective generation of bootstrap inputs H_0. Such a fictional technique may work in practice, although is also extremely difficult to implement.

In other words, the existing techniques constitute another search space: how to assemble existing techniques to achieve the best testing thoroughness within a time limit, and even limited resources in the implementation of such a technique? From a practitioner's perspective, it is extremely difficult to evaluate the effectiveness of a *single* optimization in this entire search space. Suppose that techniques A/B presented a 15%/10% coverage improvement compared with a baseline implementation. However, this does not imply that A is indeed 5% more effective than B: it is a challenge to predict what will happen when another optimization C is integrated into the system. Such a threat to validity will be intensified as more techniques are invented, and we wonder whether there is a "grand unified theory" for explaining the effectiveness of not only existing techniques but also their combined effects.

8 Related Work

Test input generation has long been considered as a search problem [23]. In the early stage of research, the focus is on the "meta-heuristics" to guide the search, and motivated research work on mutation-based $N_{\mu\chi}$ and coverage-guided neighborhood selection strategy design.

Over twenty years of technical evolution, many perspectives on automated test input generation were studied in existing survey literature: (dynamic) symbolic analysis [5, 6], genetic algorithms [11], the data flow guidance [12], or a particular application domain [13]. In contrast to these comprehensive surveys targeted at a thorough understanding of existing techniques, this paper presents the framework of $\langle N, S, H_0 \rangle$ to study the search space and search strategy of automated test input generation techniques. Under this framework, mainstream approaches (random testing, fuzzing, and DSE/symbolic analyses), long being believed to be considerably different, are unified. This new perspective offers a better chance to analytically study the characteristics, strengths, and weaknesses of existing techniques, which are generally beyond the scope of existing surveys.

The following representative fuzzing/DSE techniques are studied in this paper: (1) using taint analysis: BuzzFuzz [29], TaintScope and VUzzer [17]; (2) exploiting models: MoWF [37] and BFAFI [39]; (3) calculating logical distances: CREST [27] and FitnexFitnex [30]; (4) focusing on infrequent paths: Sub-path DSE [31] and AFLfast [9]; (5) exploiting user-provided annotations: GUIDESE [34]; (6) using textural documents: DASE [35]; (7) pruning paths: RWset [28]; (8) identifying don't care variables: Don't Care Analysis [36]; (9) merging state: VeriTesting [32] and MULTISE [33]; (10) using state progress information: Steelix [18]; (11) compositional DSE: SMART [26] and IC-Cut [38]; (12) combining random testing and DSE in an interleaving manner: Hybrid-Concolic [25]; (12) combining fuzzing and DSE in an interleaving manner: Basilisk-Framework [45], Driller [16], Mayhem [44] and QYSM [47]; (13) transforming source code: LEO [40] and T-Fuzz [49]; (14) using DSE to calculate the fuzzing mutation ratio: SYMFUZZ [43]; (15) using DSE to generate the fuzzing seed: Munch [46] and SAFL [48]; (16) using learning strategy: ParaDySE [41]. These techniques are analyzed in depth in Sections 4 and 6.

9 Conclusion

This paper presents an analysis framework to characterize an automated test input generation technique as three major components: neighborhood definition, neighborhood selection strategy, and bootstrap test inputs.

The framework provides researchers a unified perspective to investigate and understand the characteristics, strengths, and weaknesses of existing test input generation techniques: fuzzing, DSE, and their hybrids. Supported with empirical study results, the paper calls for, and sheds light on, future efforts towards more effective test input generation techniques.

Acknowledgments This work was supported in part by National Key R&D Program (Grant #2017YFB1001801) and National Natural Science Foundation (Grants #61690204 and #61802165) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

1. Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the 1999 2nd Conference on Computer Science and Engineering*, pages 21–28, 1999.
2. Bogdan Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.
3. B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, Technical Report CS-TR-1995-1268, University of Wisconsin, 1995.
4. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
5. Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic

execution for software testing in practice: Preliminary assessment. In *Proceedings of the 2011 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

6. Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, 2013.
7. Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.
8. American fuzzy lop fuzzer. `http://lcamtuf.coredump.cx/afl/technical_details.txt`.
9. Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *Acm Sigsac Conference on Computer & Communications Security*, 2016.
10. Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
11. Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. A survey on software testing techniques using genetic algorithm. *arXiv preprint arXiv:1411.1154*, 2014.
12. Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. A survey on data-flow testing. *ACM Computing Surveys*, 50(1):5, 2017.
13. Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys*, 50(5):66, 2017.
14. Joseph R. Horgan, Saul London, and Michael R Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, 1994.
15. Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2007.
16. Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, volume 16, pages 1–16, 2016.
17. Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.
18. Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.
19. Libfuzzer. `http://llvm.org/docs/LibFuzzer.html`.
20. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.
21. Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3):215–222, 1976.
22. James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
23. Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
24. Butler W Lampson and Howard E Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, 1976.
25. Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 2007 29th International Conference on Software Engineering*, pages 416–426. IEEE, 2007.
26. Patrice Godefroid. Compositional dynamic test generation. In *ACM Sigplan Notices*, volume 42, pages 47–54. ACM, 2007.
27. Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446. IEEE, 2008.
28. Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.
29. Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 2009 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.
30. Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 39th IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009.
31. You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *ACM SigPlan Notices*, volume 48, pages 19–32. ACM, 2013.
32. Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with VeriTesting. In *Proceedings of the 2014 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.
33. Koushik Sen, George Necula, Liang Gong, and Wontae Choi. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853. ACM, 2015.
34. Koushik Sen, Haruto Tanno, Xiaojing Zhang, and Takashi Hoshino. GuideSE: Annotations for guiding concolic testing. In *Proceedings of the 2015 10th International Workshop on Automation of Software Test*, pages 23–27. IEEE, 2015.
35. Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. DASE: Document-assisted symbolic execution for improving automated software testing. In *Proceedings of the 2015 37th IEEE International Conference on Software Engineering*, volume 1, pages 620–631. IEEE, 2015.
36. Cuong Nguyen, Hiroaki Yoshida, Mukul Prasad, Indradeep Ghosh, and Koushik Sen. Generating succinct test cases using don't care analysis. In *Proceedings of the 2015 8th IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015.
37. Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the*

2016 31st IEEE/ACM International Conference on Automated Software Engineering, pages 543–553. ACM, 2016.

38. Maria Christakis and Patrice Godefroid. IC-Cut: A compositional search strategy for dynamic test generation. In *Model Checking Software*, pages 300–318. Springer, 2015.
39. Hyoung Chun Kim, Young Han Choi, and Dong Hoon Lee. Efficient file fuzz testing using automated analysis of binary file format. *Journal of Systems Architecture*, 57(3):259–268, 2011.
40. Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
41. Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. Automatically generating search heuristics for concolic testing. In *Proceedings of the 2018 40th International Conference on Software Engineering*, pages 1244–1254. ACM, 2018.
42. Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):15, 2011.
43. Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 725–741. IEEE, 2015.
44. Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alex Rebert, and Ned Williamson. The Mayhem cyber reasoning system. *2018 IEEE Symposium on Security and Privacy (SP)*, 16(2):52–60, 2018.
45. Alexander Kampmann. Local analysis for global inputs. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 433–436, 2017.
46. Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. Improving function coverage with Munch: A hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 2018 33rd Annual ACM Symposium on Applied Computing*, pages 1475–1482. ACM, 2018.
47. Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.
48. Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 2018 40th International Conference on Software Engineering: Companion Proceeedings*, pages 61–64. ACM, 2018.
49. Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
50. Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. Towards optimal concolic testing. In *Proceedings of the 2018 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018.