On the Value of Backdoors for Software Engineering: a Case Study with Test Suite Generation

Jianfeng Chen, Tim Menzies
jchen37@ncsu.edu,timm@ieee.org
NC State University, USA

ABSTRACT

Within a large space of tests, many variables may have the same settings. AI researchers report that when solutions share settings, then solutions can be found, faster, by first finding and setting just a few of the shared variables- a phenomenon they call "backdoors".

This paper shows that backdoors can dramatically improve test case generation. If backdoors exist, then they exist in most solutions. Hence, test suites can be quickly generated just by sampling around average values seen in a few randomly selected valid tests (as done by our new algorithm called Snap).

When applied to 27 real-world test case studies Snap ran 10 to 3000 times faster (median to max) than a prior report (at ICSE'18). While prior work found tests that were 70% valid, all Snap's tests were valid Test engineers would find it easier to use Snap's tests since those test suites are 10 to 750 times smaller (median to max) than those found by prior work.

KEYWORDS

SAT solvers, test suite generation, mutation

1 INTRODUCTION

Many software testing or verification problems (such as test suite minimization, combinatorial testing, and test case prioritization) can be transformed into "SAT"; i.e. a propositional satisfiability problem (see §2.2 for details). In theory, this is a useful transformation since "SAT solvers" are well-studied algorithms in computer science. As Micheal Lowry said at a panel at ASE'15:

> *"It used to be that reduction to SAT proved a problem's intractability. But with the new SAT solvers, that reduction now demonstrates practicality."*

However, in practice, general SAT solvers, such as the Z3 [14], MathSAT [8], vZ [6] *et al.*, are challenged by the complexity of real-world software models. For example, the largest benchmark for SAT Competition 2017 [25] had 58,000 variables– which is far smaller than (e.g.) the 300,000 variable problems seen in the recent SE testing literature [15]. Accordingly, researchers explore various heuristics to take best advantage of the SAT solvers. For example Dutra *et al.* argued at ICSE'18 [15] that test variants built from other valid tests are probably also valid. In their approach, new tests are generated using the deltas between valid cases a, b, c:

$$d = c \oplus (a \oplus b) \tag{1}$$

This is a useful heuristic for speeding up test case generation since the $\oplus$ "exclusive or" operator is much faster to apply than a theorem prover. But heuristics like Eq. 1 must be applied cautiously. In their haste to quickly generate solutions, these heuristics might introduce new problems. For example, in the case of *QuickSampler*, algorithm-generated million of samples without validation guarantees. Such invalid solutions cannot be applied during the testing. Hence, in practice, the results from a heuristic test generation may require a post-generation "sanity check". In the case of *QuickSampler*, this sanity check would take more than 50 hours (i.e. much longer than the original execution time). Another issue with heuristic methods is that they may not always generate unique tests. For example, in one sample of 10 million tests generated from the *blasted_case47* benchmark. *QuickSampler* only found 26,000 unique valid solutions. That is, 99% of the tests were repeating other tests.

It turns out that repeated solutions can be exploited, to great effect. When used for test generation, SAT solvers represent software code as propositional formula (using the methods of §2.2). AI researchers report that when the solutions to propositional formula share settings, then solutions can be found, faster, by first finding and setting some of the shared variables [42]. AI researchers call this phenomenon the "**backdoor effect**" [42].

The trick to exploiting backdoors is finding the "backdoor variables"; i.e. those with settings shared by most solutions. The starting point for this paper was the observation that, when such backdoors exist, they can be seen in the common settings of a random sample of successful examples (in our case, valid test cases). We hence conjectured that, to exploit backdoors for test generation, just:

> *Sample around the average values seen in a few randomly selected valid tests.*

We call such exploration the "Snap tactic". The intuition here is that, when backdoors exist, the common settings seen in a small M sample may also be the common settings in a large $N \gg M$ sample.

To evaluate this intuition we implemented the tactic in an algorithm called Snap that combines Eq. 1 with the Z3 theorem prover with the Snap tactic (i.e. sample around average values seen in a few randomly selected valid tests). Then we study four questions:

RQ1: How reliable is the Eq. 1 heuristic? One reason we advocate Snap is that this method fully verifies each test. But is that necessary? How often does Eq. 1 produce invalid tests? Our experiments confirm Dutra *et al.*'s estimate that the percent of invalid tests generated by Eq. 1 is 30% or less. But that median result does not fully characterize the variability of that distribution. In a third of case studies, the percent of valid tests generated by Eq. 1 is very low (25% to 50%) (median to max). Hence:

> ***Conclusion #1:*** Eq. 1 should not be used without verification of the resulting test.

In this regard, it is significant to say that since Snap's tests suites are so small, it is possible to quickly verify all tests. That is, unlike *QuickSampler*, all Snap's tests are valid.

Submitted to ICSE'2020, August 23, 2019.

RQ2: How fast is Snap? In a result confirming the value of backdoors for test suite generation, we see that:

> ***Conclusion #2:*** Snap was 10 to 3000 times faster than *QuickSampler* (median to max).

RQ3: How easy is it to apply Snap's test cases? Pragmatically, the *smaller* a test suite, the *easier* it is for programmers to run those tests. We find that:

> ***Conclusion #3:*** Snap's test cases were 10 to 750 times smaller than those of *QuickSampler* (median to max).

Small tests suites are important since:

- Industrial researchers [45] advise that one of the major costs of testing is the developer time required to investigate failed tests. If we are running 10 to 750 fewer tests, then that also reduces how much developer time is spent on managing failed tests.
- When test suites are 10 to 750 times smaller, then they are faster to execute. Faster test execution means that software teams can certify a new release, quicker. This is particularly important for organizations using continuous integration since faster test suites mean they can make more releases each day– which means that clients can sooner receive new (or fixed) features.
- Many current organizations spend tens of millions of dollars each year (or more) on cloud-based facilities to run large tests suites [45]. The fewer the tests those organizations have to run, the cheaper their testing.
- Finally, the methods of this paper automate the difficult task of finding test suite *inputs*, with the goal of driving the tests deep within many parts of the software. But what *outputs* are required/ expected/ deprecated when those inputs are given to a program? If testing for (e.g.) core dumps, then specifying off-nominal behavior is trivial (just look for a core dump file). But for many other, more nuanced, business cases, specifying what should (and should not) be seen when a test executes is a time-consuming task requiring a deep understanding of the purpose and context of the software. By minimizing the number of tests being executed, we are also minimizing the developer effort required to specify the expected nominal and off-behavior associated with each test execution.

RQ4: How diverse are the Snap test cases? Since Snap explores far fewer tests than *QuickSampler*, its tests suites could be less diverse. Nevertheless:

> ***Conclusion #4:*** The diversity of Snap's test suites is very similar to those of *QuickSampler*.

In summary, the unique contributions of this paper are:

- To the best of our knowledge, this paper is the first application in SE testing of backdoor-based reasoning.
- Another contribution of this paper is the definition and evaluation of several techniques for exploiting the backdoor variables (using "sample the average values of some valid solutions").
- This paper offers an open-source version of a tool called Snap that encodes these backdoor exploitation methods[1]. While this is more of a "systems" contribution than a "research" contribution, the availability of such a reference system lets other researchers to check and extend our results.
- We show that for test-case generation, backdoor-based reasoning generates much smaller solutions as the prior state-of-the-art; and does so far faster. Further, 100% of our tests are valid (while other methods may only generate 70% valid tests, or less).

The rest of this paper is structured as follows. The next section offers background notes on backdoor-based reasoning; and using SAT solvers for test case generation. After that, we describe methods for quickly finding backdoors. These are then tested on the same case studies used in the ICSE'18 *QuickSampler* paper. After showing those empirical results, we discuss threats to validity.

Before starting, we digress to make the following point. This paper should not be read as a criticism of *QuickSampler*. Rather, our aim is to explore core computational processes (tactics for the faster exploration of propositional equations) within SE. We explore test case generation since that is a hard problem of much relevance to current SE practice (see the last point). For that exploration, we use *QuickSampler* as a baseline reference system from the recent SE testing literature. Such baselines are important since, using it, we can then comparatively assess other methods (e.g. Snap's backdoor-based reasoning). In this case, the results of that assessment were quite dramatic: orders of magnitude reduction in (a) test generation time as well as (b) the size of the generated tests; also (c) 100% of our tests are valid.

2 BACKGROUND

2.1 About Backdoors

As described in the next section, generating software tests can be characterized as solving a propositional formula. This is a useful insight since propositional satisfiability is a well-studied problem with many mature tools. Modern constraint solvers, i.e. SAT-solvers are based on so-called Davis-Putnam-Logemann-Loveland (DPLL) procedure [13], or some variant. The DPLL procedure searches systematically for a satisfying assignment, applying first unit propagation and pure literal elimination as often as possible. Then, DPLL branches on the truth value of a variable, and recurses.

SAT has applications in many areas, including areas outside of software engineering. For this reason, the AI literature contains numerous studies on SAT solvers. That literature has lead to some surprising results. For example, Williams *et al.* [42] defined "backdoors" as small sets of variables that capture the overall combinatorics of a problem. Later Gaspers *et al.* [22] formally defined the "backdoor" as

> *A backdoor set is a set of variables of a propositional formula such that fixing the truth values of the variables in the backdoor set moves the formula into some polynomial-time decidable class.*

[1] Source code at http://github.com/blindedForReview

```
1 int mid(int x, int y, int z) {
2  if (x < y) {
3    if (y < z) return y;
4    else if (x < z) return z;
5    else return x;
6  } else if (x < z) return x;
7  else if (y < z) return z;
8  else return y; }
```

The code above has the six branches shown below. Each branch is a logical constraint $C_1 \vee C_2 \vee C_3 \ldots \vee C_6$. A valid test selects `x, y, z` such that it satisfies these constraints.

```
path 1: [C1: x < y < z] L2->L3
path 2: [C2: x < z < y] L2->L3->L4
path 3: [C3: z < x < y] L2->L3->L4->L5
path 4: [C4: y < x < z] L2->L6
path 5: [C5: y < z < x] L2->L6->L7
path 6: [C6: z < y < x] L2->L6->L7->L8
```

via SMT conversion tools [20]. By convention, the disjunction $\vee C_i$ is transformed into the conjunction normal form (CNF) $C'_1 \wedge C'_2 \ldots$. A valid assignment to the CNF, i.e. the assignment that fulfills all clauses, is corresponding to a test case, covering some branch of code.

Figure 1: A script of C programming can be translated into CNF form, the target problem discussed in this paper.

That is, backdoors are a way to turn very slow exponential time tasks into much faster polynomial-time tasks. Numerous studies have confirmed the existence of such backdoors, in many domains[2, 4, 31, 36]. Fichte *et al.* [19] proposed backdoor-based methods to solve the answer set programming problem (ASP), i.e. the search for answer sets in disjunctive logic programs. Up to then, the most successful solvers for disjunctive ASP [23] were based on SAT techniques and the concept of loop formulas [30]. Fichte *et al.* exploited the small distance of a disjunctive program from being normal (such distance is measured in terms of the size the smallest backdoor to normality, i.e. the smallest number of atoms whose deletion makes the program normal). Among the structured (i.e. non-random) domains, experiments showed that only 1 to 10% of atoms were included in the smallest strong backdoors. Also, Kronegger *et al.* [29] proposed an algorithm to filter the backdoors for *planning* problem. Their "backdoors for planning" method built upon the so-called causal graph. Such graphs model the dependencies between variables in planning instances. Based on the structure of the causal graph, various tractable fragments of planning can be identified. The algorithm contains two phases: 1) the detection phase – finding the small backdoor and 2) the evaluation phase – using the additional information given in the backdoor to solve the planning instance.

The above works show that a small ratio ($< 10\%$) of variables, i.e. the backdoors, did significantly reduce the complexity of the SE models. But showing that the backdoors exist is a different question to "how to find the backdoors, quickly". Fichte and Kronegger etc.'s methods are not optimized to reduce CPU cost or minimize the size of the returned model (which are two properties that we desire in test suites). What's more, their algorithms are not simple to understand and implement, due to the somewhat arcane and tricky nature of their lemmas and propositions.

2.2 Theorem Proving in Software Testing

This section describes some of the ways software testing can be recast as a theorem proving problem. Note that, once recast in this way, SAT solvers can be used as test case generation tools.

Arito *et al.* proposed a framework to transform the test suite minimization problem (TSMP) in regression testing into a constrained SAT problem [3]. This transformation is done by modeling TSMP instances as a set of Pseudo-Boolean constraints that are later translated to SAT instances. TSMP has two objectives: 1) minimizing the testing cost and 2) maximizing the program coverage. To start with, a set of test cases $\mathcal{T} = \{t_1, t_2, t_3, \ldots\}$ as well as their running time cost $\{c_1, c_2, \ldots\}$ is defined.

- t_i is a binary signal indicating if the test case i should be tested.
- The information about whether test case t_i covers some element in the program e_j is stored as the binary matrix $M = [m_{ij}]$.

To translate the TSMP into constrained problems, we have all following pseudo-boolean constraints: $\sum_{i=1}^{n} c_i t_i \leq B$ and $\sum_{j=1}^{m} e_j \geq P$ where $B \in \mathbb{Z}$ is the maximum allowed cost and $P \in \{1, 2, \ldots, m\}$ is the minimum coverage level. Having the pseudo-boolean constraints, Een *et al.* [16] provides three techniques to translate pseudo-boolean constraints (linear constraints over boolean variables) into clauses that can be handled by a SAT-solver.

As another scenario, combinatorial testing [38] can be expressed in the form of conjunctive normal form (CNF) and hence studied by a SAT solver. Combinatorial testing covers interactions of parameters in the system under test. A well-chosen sampling mechanism can reduce the cost of software and system testing by reducing the number of test cases to be executed [44]. Considering the system environment {AMD, Intel} × {Windows, MacOS, Linux} × {IE, Firefox, Safari}. Note that not all combinations valid. For example, MacOS does not support AMD processor while IE does not support MacOS, etc. All of such constraints can be expressed as the feature model [27] or as product lines. Further, such a feature model can be transformed into the CNF formulas [35], at which point, SAT solvers can compute out the valid testing environment combination.

Last but not least, given a script of C programming, one can translate it into CNF formulas, as done in Figure 1. Symbolic/dynamic execution techniques [5, 12] extract the possible execution branches of a procedural program. Each branch is a conjunction of conditions $B_i = C_x \wedge C_y \wedge \ldots$ so the whole program can be summarized as the disjunction $B_i \vee B_j \vee \ldots$. Using deMorgan's rules[2] these clauses can be converted to conjunctive normal form (CNF) where the inputs to the program are the variables in the CNF.

[2] Disjunctions to conjunctions: $P \vee Q \equiv (\neg P \wedge \neg Q)$
Conjunctions to disjunctions: $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$.

Table 1: SNAP and its related work for solving theorem proving constraints via sampling.

Reference	Year	Citation	Sampling methodology	Case study size (max|variables|)	Verifying samples	Distribution/ diversity reported
[47]	1999	105	Binary Decision Diagram	≈1.3K	○	○
[26]	2003	50	Interval-propagation-based	200	○	○
[46]	2004	54	Binary Decision Diagram	< 1K	○	○
[41]	2004	141	Random Walk + WalkSat	*No experiment conducted*		
[24]	2011	88	Sampling via determinism	6k	○	○
[17]	2012	25	MaxSat + Search Tree	*Experiment details not reported*		
[11]	2014	29	Hashing based	400K	○	◑
[9]	2015	28	Hashing based (paralleling)	400K	○	◑
[34]	2016	29	Universal hashing	400K	○	◑
[15]	2018	5	Z3 + Eq. 1 flipping	400K	○	◑
SNAP	2019	this paper	Z3 + Eq. 1 + local sampling	400K	●	●

○ / ● : the absence / presence of corresponding item ◑ : only partial case studies *(the small case studies)* were reported

2.3 Theorem Prover Research in SE

As shown in Table 1, much prior research has explored scaling theorem proving for software engineering. One way to tame the theorem proving problem is to simplify or decompose the CNF formulas. A recent example in this arena was *GreenTire*, proposed by Jia *et al.* [28]. *GreenTire* supports constraint reuse based on the logical implication relation among constraints. One advantage of this approach is its efficiency guarantees. Similar to the analytical methods in linear programming, they are always applied to a specific class of problem. However, even with the improved theorem prover, such methods may be difficult to be adopted in large models. *GreenTire* was tested in 7 case studies. Each case study was corresponding to a small code script with ten lines of code, e.g. the *BinTree* in [40]. For the larger models, such as those explored in this paper, the following methods might do better.

Another approach, which we will call sampling, is to combine theorem provers Z3 with stochastic sampling heuristics. For example, given random selections for b, c, Eq. 1 might be used to generate a new test suite, without calling a theorem prover. Theorem proving might then be applied to some (small) subset of the newly generated tests, just to assess how well the heuristics are working.

The earliest sampling tools were based on binary decision diagrams (BDDs) [1]. Yuan *et al.* [46, 47] build a BDD from the input constraint model and then weighted the branches of the vertices in the tree such that a stochastic walk from root to the leaf was able to generate samples with the desired distribution. In other work, Iyer proposed a technique named *RACE* which has been applied in multiple industrial solutions [26]. *RACE* (a) builds a high-level model to represent the constraints; then (b) implements a branch-and-bound algorithm for sampling diverse solutions. The advantage of *RACE* is its implementation simplicity. However, *RACE*, as well as the BDD-based approached introduced above, return highly biased samples, that is, highly non-uniform samples. For testing, this is not recommended since it means small parts of the code get explored at a much higher frequency than others.

Using a SAT solver *WalkSat* [39], Wei *et al.* [41] proposed *SampleSAT*. *SampleSAT* combines random walk steps with greedy steps from *WalkSat*– a method that works well for small models. However, due to the greedy nature of *WalkSat*, the performance of *SampleSAT* is highly skewed as the size of the constraint model increases.

For seeking diverse samples, universal hashing [33] techniques have been proposed. These algorithms were designed for strong guarantees of uniformity. Meel *et al.* [34] provided an overview of key ingredients of integration of universal hashing and SAT solvers; e.g. with universal hashing, it is possible to guarantee uniform solutions to a constraint model. These hashing algorithms can be applied to the extreme large models (with near 0.5M variables). More recently, several improved hashing-based techniques have been purposed to balance the scalability of the algorithm as well as diversity (i.e. uniform distribution) requirements. For example, Chakraborty *et al.* proposed an algorithm named *UniGen* [11], following by the *Unigen2* [9]. *UniGen* provides strong theoretical guarantees on the uniformity of generated solutions and has applied to constraint models with hundreds of thousands of variables. However, *UniGen* suffered from a large computation resource requirement. Later work explored a parallel version of this approach. *Unigen2* achieved near linear speedup of the number of CPU cores.

To the best of our knowledge, the state-of-the-art technique for generating test cases using theorem provers is *QuickSampler* [15]. *QuickSampler* was evaluated on large real-world case studies, some of which have more than 400K variables. At ICSE'18, it was shown that *QuickSampler* outperforms aforementioned *Unigen2* as well as another similar technique named *SearchTreeSampler* [17]. *QuickSampler* starts from a set of valid solutions generated by Z3. Next, it computes the differences between the solutions using Eq. 1. New test cases generated in this manner are not guaranteed to be valid. *QuickSampler* defines three terms, we use later in this paper:

- A test suite is a set of valid tests.
- A test is *valid* if uses input settings that satisfy the CNF.
- One test suite is more *diverse* than another if it uses more variable within the CNF disjunctions. *Diverse* test suites are preferred since they cover more parts of the code.

According to Dutra *et al.*'s experiments, the percent of valid tests found by *QuickSampler* can be higher than 70%. The percent of

0) **Set up**
   (a) let $N = 100$; i.e. initial sample size;
   (b) let $k = 5$; i.e. number of clusters;
   (c) let *suite*= $\emptyset$; i.e. the output test suite;
   (d) let *samples*= $\emptyset$; i.e. a temporary work space.
1) **Initial samples generation**:
   (a) Add N solutions (from Z3) to *samples*
   (b) Put all *samples* into *suite* (since they are valid)
2) **Delta preparation**:
   (a) Find delta $\delta = (a \oplus b)$ for all $a, b \in$ *samples*.
   (b) Weight each delta by how often it repeats
3) **Sampling**
   (a) Find k centroids in *samples* using k-means ;
   (b) For each centroid c, repeat N times:
      (i) Stochastically pick deltas δ_i, δ_j at prob. equal to their weight.
      (ii) Get new candidate via $c \oplus (\delta_i \vee \delta_j)$
      (iii) Verify new candidate using Z3;
      (iv) If invalid, repair using Z3 (see §3.1). Add to *sample*;
      (v) Add new to *suite*;
4) **Loop or terminate**:
   (a) If diversity improving (see §3.2), go to step 2. Else return *suite*.

Figure 2: Snap

valid tests found by Snap, on the other hand, is 100%. Further, as shown below, Snap builds those tests with enough diversity much faster than *QuickSampler*.

3 IMPLEMENTING THE SNAP TACTIC

In the Snap algorithm if Figure 2, each test is a set of zeros or ones (false, true) assigned to all the variables in a CNF formula.

As shown in **initial samples** (steps 1a,1b), instead of computing some deltas between many tests, Snap restrains mutation to the deltas between a few valid tests (generated from Z3). Snap builds a pool of 10,000 deltas from $N = 100$ valid tests (which mean calling a theorem prover only $N = 100$ times). Snap uses this pool as a set of candidate "mutators" for existing tests (and by "mutator", we mean an operation that converts an existing test into a new one).

After that, in **delta preparation** (steps 2a,2b), Snap applies Eq. 1. Note that the more often a setting repeats, the more likely it is a backdoor variable. Hence, step 2b sorts the deltas on occurrence frequency. This sort is used in step 3b.

In **sample** (steps 3a,3b), Snaps samples around the average values seen in a few randomly selected valid tests. Here, "averaging" is inferred by using the median values seen in k clusters. Note that, in step 3b, we use deltas that are more likely to be backdoor variables (i.e. we use the deltas that occur more frequently).

Step 3b.ii is where we verify the new candidate using Z3. Snap explores far fewer candidates than *QuickSampler* (10 to 750 times fewer, see §5.3). Since we are exploring less, we can take the time to verify them all. Hence, 100% of Snap's tests are valid (and the same is *not* true for *QuickSampler*– see §5.1).

Note that in 3b.iv, if a candidate passes verification, we output it then forget about it. Else, we repair it and add it to our clusters. We do this since test cases that pass verification do not add new information to our samples. However, when an instance fails verification and is repaired, that offers new settings.

Note also that Snap takes great care in how it calls a theorem prover. Theorem provers are much slower for *generating* new tests than *repairing* invalid tests than for *verifying* that a test is valid (since there are more options for generation that for repairing than for verification). Hence, Snap needs to *verify* more than it *repairs* (and also do *repairs* more than *generating* new tests). Accordingly,

The call to Z3 in step 1a can be the slowest (since this a *generate* call that must navigate all the constraints of our CNF). Hence, we only do this $N = 100$ times. Also, the call to Z3 in step 3b.iii is a *verification call* and is much faster since all the variables are set. Finally, The call of Z3 in the step 3b.iv *repair* call, is slower than step 3b.iii since (as discussed below), our repair operator introduces some open choices into the test. Note that we only need to repair the small minority of new tests that fail verification. Later in this paper, we can use Figure 4 to show that repairs are only needed on 30% (median) of all tests.

3.1 Implementing "Repair"

Snap's repair function deletes "dubious" parts of a test case, then uses Z3 to fill in the the gaps. In this way, when we repair a test, most bits are set and Z3 only has to search a small space.

To find the "dubious" section, we reflect on how step 3b.ii operates. Recall that the new test uses $\delta = a \oplus b$ and a, b are valid tests taken from *samples*. Since a, b were valid, then the "dubious" parts of the test is anything that was not seen in both a and b. Hence, we preserve the bits in $c \oplus \delta$ bits (where the corresponding δ bit was 1), while removing all other bits (where δ bit was 0). For example:

- When mutating c =(1,0,0,1,1,0,0,0) using δ =(1,0,1,0, 1,0,1,0).
- If $c \oplus \delta$ =(0,0,1,1,0,0,1,0) is invalid, then Snap deletes the "dubious" sections as follows.
- Snap preserves any "1" bits that were seen in δ.
- Snap deletes the others; e.g. bits 2, 4, 6, 8 (0,~~0~~,1,~~1~~,0,~~0~~, 1,~~0~~).
- Z3 is then called to fill out the missing bits of (0?1?0?1?).

3.2 Implementing "Termination"

To implement Snap's termination criteria (step 4a), we need a working measure of diversity. Recall from the introduction that one test suite is more *diverse* than another if it uses more of the variable settings with disjunctions inside the CNF. *Diverse* test suites are *better* since they cover more parts of the code.

To measure diversity, we used Feldt *et al.* [18]'s normalized compression distance (NCD). A test suite with high NCD implies higher code coverage during the testing[3]. NCD uses `gzip` to the estimate Kolmogorov complexity [32] of the tests. If $C(x)$ is the length of compression of x and $C(X)$ is the compression length of binary string set X's concatenation, then:

$$\mathrm{NCD}(X) = \frac{C(X) - \min_{x \in X}\{C(x)\}}{\max_{x \in X}\{C(X \backslash \{x\})\}} \quad (2)$$

Snap exits if NCD improves by $X \leq 5\%$ in the last $T = 10$ minutes.

3.3 Engineering Choices

Snap uses theses control parameters (set via engineering judgment):

[3]Aside: we note that we did not adopt the diversity metric (distribution of samples displayed as a histogram) from [9, 15] since computing that metric is very time-consuming. For the case studies of this paper, that calculation required days of CPU.

- $X = 5\%$;
- $T = 10$ minutes;
- $N = 100$ samples;
- $k = 5$ clusters.

In future work, it could be insightful to vary these values.

Another area that might bear further investigation is the clustering method used in step 3a. For this paper, we tried different clustering methods. Clustering ran so fast that we were not motivated to explore alternate algorithms. Also, we found that the details of the clustering were less important than pruning away most of the items within each cluster (so that we only mutate the centroid).

4 EXPERIMENTAL SET-UP

4.1 Code

To explore the research questions shown in the introduction, the Snap system shown in Algorithm 2 was implemented in C++ using Z3 v4.8.4 (the latest release when the experiment was conducted). A k-means cluster was added using the free edition of ALGLIB [7], a numerical analysis and data processing library delivered for free under GPL or Personal/Academic license. *QuickSampler* does not integrate the samples verification into the workflow. Hence, in the experiment, we adjusted the workflow of *QuickSampler* so that all samples are verified before termination. Also, the outputs of *QuickSampler* were the assignments of independent support. The *independent support* is a subset of variables which completely determines all the assignments to a formula [15]. In practice, engineers need the complete test case input; consequently, for valid samples, we extended the *QuickSampler* to get full assignments of all variables from independent support's assignment via propagation.

4.2 Experimental Rig

We compared Snap to the state-of-the-art *QuickSampler*, technique purposed by Dutra *et al.* at ICSE'18. To ensure a repeatable result, we updated the Z3 solver in *QuickSampler* into the latest version.

To reduce the observation error and test the performance robustness, we repeated all experiment 30 times with 30 different random seeds. To simulate real practice, such random seeds were used in Z3 solver (for initial solution generation), ALGLIB (for the k-means) and other components. Due to the space limitation, we cannot report results for all 30 repeats. Correspondingly we report the medium or the IQR (75-25th variations) results.

All experiments were conducted on Xeon-E5@2GHz machines with 4GB memory, running CentOS. These were multi-core machines but for systems reasons, we only used one core per machine.

4.3 Case Studies

Table 2 lists the case studies used in this work. We can see that the number of variables ranges from hundreds to more than 486K. The large examples have more than 50K clauses, which is very huge. For exposition purposes, we divided the case studies into three groups: the small case studies with vars $< 6K$; the medium case studies with $6K <$ vars $< 12K$ and the large case studies with vars $> 12K$.

For the following reasons, our case studies are the same as those used in the *QuickSampler* paper:

Table 2: Case studies used in this paper. Sorted by number of variables. Medium sized-problems are highlighted with blue rows while the large ones are in orange rows. Three items (marked with *) are not included in some further reports (see text). See text for details.

Size	Case studies	Vars	Clauses
Small	blasted_case47	118	328
	blasted_case110	287	1263
	s820a_7_4	616	1703
	s820a_15_7	685	1987
	s1238a_3_2	685	1850
	s1196a_3_2	689	1805
	s832a_15_7	693	2017
	blasted_case_1_b12_2*	827	2725
	blasted_squaring16*	1627	5835
	blasted_squaring7*	1628	5837
	70.sk_3_40	4669	15864
	ProcessBean.sk_8_64	4767	14458
	56.sk_6_38	4836	17828
	35.sk_3_52	4894	10547
	80.sk_2_48	4963	17060
Medium	7.sk_4_50	6674	24816
	doublyLinkedList.sk_8_37	6889	26918
	19.sk_3_48	6984	23867
	29.sk_3_45	8857	31557
	isolateRightmost.sk_7_481	10024	35275
	17.sk_3_45	10081	27056
	81.sk_5_51	10764	38006
	LoginService2.sk_23_36	11510	41411
Large	sort.sk_8_52	12124	49611
	parity.sk_11_11	13115	47506
	77.sk_3_44	14524	27573
	20.sk_1_51	15465	60994
	enqueueSeqSK.sk_10_42	16465	58515
	karatsuba.sk_7_41	19593	82417
	tutorial3.sk_4_31	486193	2598178

- We wanted to compare our method to *QuickSampler*;
- Their case studies were online available;
- Their case studies are used in multiple papers [9, 11, 15, 34] etc.

These case studies are representative of scenarios engineers met in software testing or circuit testing in embedded system design. They include bit-blasted versions of SMTLib case studies, ISCAS89 circuits augmented with parity conditions on randomly chosen subsets of outputs and next-state variables, problems arising from automated program synthesis and constraints arising in bounded theorem proving. For more introduction of the case studies, please see [9, 15].

For pragmatic reasons, certain case studies were omitted from our study. For example, we do not report on *diagStencilClean.sk_41_36* in the experiment since the purpose of this paper is to sample a set of valid solutions to meet the diversity requirement; while there are only 13 valid solutions from this model. The *QuickSampler* spent 20 minutes (on average) to search for one solution.

Also, we do report on the case studies marked with a star(*) in Table 2. Based on the experiment, we found that even though the

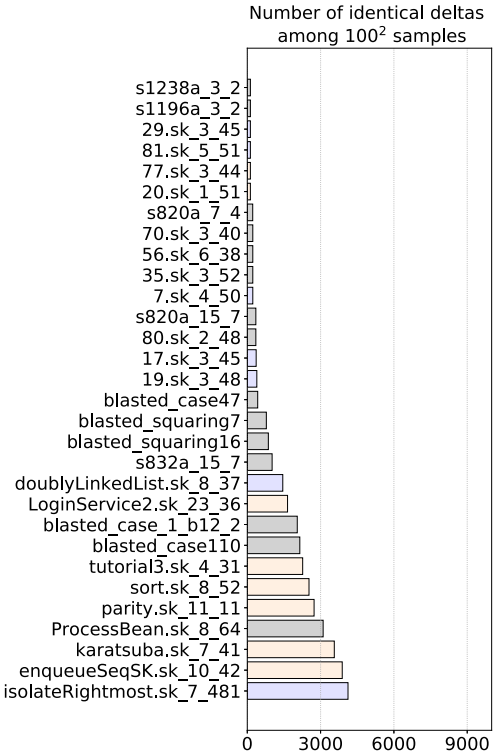

Figure 3: Number of identical deltas among 100*100 pair of valid solution deltas for all case studies. Same color scheme as Table 2.

QuickSampler generates tens of millions of samples for these examples, all samples were the assignment to the *independent support* (defined in §4.1). The omission of these case studies is not a critical issue. Solving or sampling these examples is not difficult; since they are all very small, as compared to other larger case studies.

5 RESULTS

The rest of this paper use the machinery defined above to answer the four research questions posed in the introduction. Before answering those questions, we offer one negative result. Table 2 color-coded our case studies such that the medium to large case studies is shown in blue and orange. Looking across all the following results, with the exception of runtimes, we see no pattern in the color coding (e.g. it is *not* true that larger problems have more duplicates). From this lack-of-pattern, we conclude that the complexity of test suite generation comes from the relationships between the variables, and not necessarily the number of variables themselves.

5.1 RQ1: How Reliable is the Eq. 1 Heuristic?

QuickSampler ran quickly since it assumed that tests generated using Eq. 1 did not need verification. To check that assumption, for each case study, we randomly generated 100 valid solutions, $S = \{s_1, s_2, \ldots s_{100}\}$ using Z3. Next, we selected three $\{a, b, c\} \in S$ and built a new test case using Eq. 1; i.e. $new = c \oplus (a \oplus b)$.

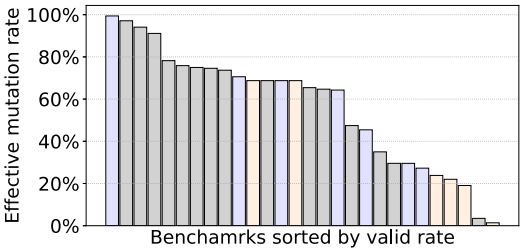

Figure 4: RQ1 results: percentage of valid mutations found it step3b.iii (computed separately for each case study).

Figure 3 lists the number of identical deltas seen in 100^2 of those deltas. Among all case studies, we rarely found large sets of unique deltas. Hence, among the 100 valid solutions given by Z3, many δs were shared within pairwise solutions. This is important since if otherwise, the Eq. 1 heuristic would be dubious.

The percentage of these deltas that proved to be valid in step3b.iii of Algorithm 1 are shown in Figure 4. Dutra *et al.*'s estimate was that the percentage of valid tests generated by Eq. 1 was usually 70% or more. As shown by the median values of Figure 4, this was indeed the case. However, we also see that in the lower third of those results, the percent of valid tests generated by Eq. 1 is very low: 25% to 50% (median to max). This result alone would be enough to make us cautious about using *QuickSampler* since, when the Eq. 1 heuristics fails, it seems to fail very badly. We recommend:

> ***Conclusion #1:*** Eq. 1 should not be used without verification of the resulting test.

By way of comparisons, it is useful to add here that SNAP verifies every test case it generates. This is practical for SNAP, but impractical for *QuickSampler* since these two systems typically process 10^2 to 10^8 test cases, respectively. In any case, another reason to recommend SNAP is that this tool delivers tests suites where 100% of all tests are valid.

5.2 RQ2: How Fast is SNAP?

Figure 5 shows the execution time required for SNAP and *QuickSampler*. The y-axis of this plot is a log-scale and shows time in seconds. These results are shown in the same order as Table 2. That is, from left to right, these case studies grow from around 300 to around 3,000,000 clauses.

For the smaller case studies, shown on the left, SNAP is sometimes slower than *QuickSampler*. Moving left to right, from smaller to larger case studies, it can be seen that SNAP often terminates much faster than *QuickSampler*. On the very right-hand side of Figure 5, there are some results where is seems SNAP is not particularly fastest. This is due to the log-scale applied to the y-axis. Even in these cases, SNAP is terminating in less than our while other approaches need more than two hours.

Figure 6 is a summary of Figure 5 that divides the execution time for both systems. From this figure it can be seen:

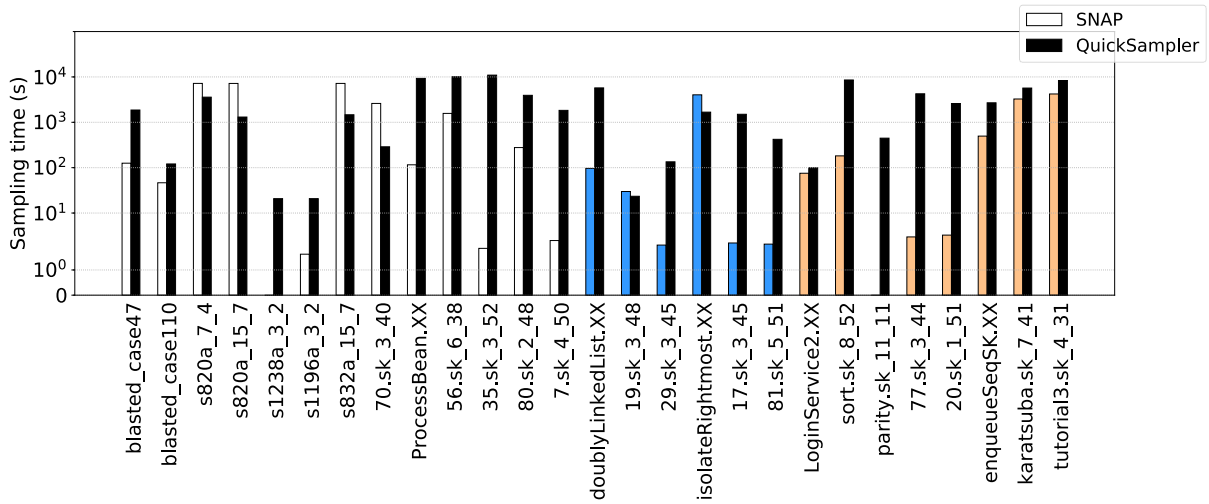

Figure 5: RQ3 results: Time to terminated (seconds), The y-axis is in log scale. The Snap sampling time for *s1238_a_3_2* and *parity.sk_11_11* is not reported since their achieved NCD were much worse than *QuickSampler*'s (see Figure 7). Figure 6 illustrates the corresponding speedups.

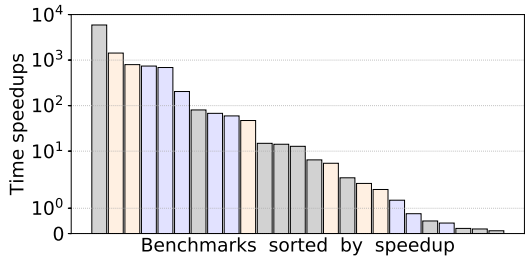

Figure 6: RQ2 results: Sorted *speedup* (time(*QuickSampler*) / time(Snap)). If over 10^0, then Snap terminates earlier.

> ***Conclusion #2:*** Snap was 10 to 3000 times faster than *QuickSampler* (median to max).

There are some exceptions to this conclusion, where *QuickSampler* was faster than Snap (see the right-hand-side of Figure 6). Those cases are usually for small models (17,000 clauses or less). For medium to larger models, with 20,000 to 2.5 million clauses, Snap is often orders of magnitude faster.

5.3 RQ3: How Easy is it to Apply Snap's Test Cases?

Table 3 compares the number of tests from *QuickSampler* and Snap. As shown by the last column in that table:

> ***Conclusion #4:*** Snap's test cases were 10 to 750 times smaller than those of *QuickSampler* (median to max).

Hence we say that using Snap is easier than other methods, where "easier" is defined as per our *Introduction*. That is, when test suites are 10 to 750 times smaller, then they are faster to run, consumes less cloud-compute resources, and means developers have to spend less time processing failed tests.

5.4 RQ4: How Diverse are the Snap Test Cases?

Figure 7 compares the diversity of the test suites generated by our two systems. These results are expressed as ratios of the observed NCD values. Results less than one indicate that Snap's test suites are less diverse than *QuickSampler*. In the median case, the ratio is one; i.e. in terms of central tendency, there is no difference between the two algorithms.

We have analyzed the Figure 7 results with a bootstrap test at 95% confidence (to test for statistically significant results), and a Cohen's effect size test (to rule out trivially small differences). Based on those tests, we say that in $\frac{20}{25} = 80\%$ of these results, there is no significant difference (of non-trivial size) between the two algorithms. Further, in two cases where there was a statistical difference (tutorial2,sk_4_31 and karatsuba_sk_7_41) the difference is less than 10%. Pragmatically, we argue that such a small difference is not troubling. Hence, we say:

> ***Conclusion #4:*** The diversity of Snap's test suites is very similar to those of *QuickSampler*.

That said, there are two examples where Snaps's diversity is markedly less than one (see s1238a_3_2 and parity.sk_11_11). In terms of scoring different algorithms, it could be argued that these examples might mean that *QuickSampler* is the preferred algorithm but only (a) if numerous invalid tests are not an issue; (b) if testing resources are fast and cheap (so saving time and money on cloud-compute test facilities is not worthwhile); and (c) if developer time is cheap (so the time required to specify expected test output, or processing large numbers of failed tests, is not an issue).

Table 3: RQ3: results. Number of unique valid cases in test suite. Sorted by last column. Same color scheme as Table 2.

Case studies	S_S SNAP	S_Q *QuickSampler*	S_Q/S_S
blasted_case47	2899	71	0.02
isolateRightmost	15480	7510	0.49
LoginService2	404	210	0.52
19.sk_3_48	204	200	0.98
70.sk_3_40	3050	4270	1.40
s820a_15_7	29065	70099	2.41
29.sk_3_45	225	660	2.93
s820a_7_4	37463	124457	3.32
s832a_15_7	27540	96764	3.51
s1196a_3_2	225	1890	8.40
enqueueSeqSK	338	2495	7.38
blasted_case110	274	2386	8.71
tutorial3.sk_4_31	336	2953	8.79
81.sk_5_51	227	2814	12.40
sort.sk_8_52	812	10184	12.54
karatsuba.sk_7_41	139	4210	30.29
20.sk_1_51	239	10039	42.00
doublyLinkedList	278	12042	43.32
17.sk_3_45	228	12780	56.05
ProcessBean	1193	75392	63.20
7.sk_4_50	258	18090	70.12
56.sk_6_38	1827	149031	81.57
80.sk_2_48	653	54440	83.37
77.sk_3_44	245	33858	138.20
35.sk_3_52	258	193920	751.63

We note that such an argument is orthogonal to the goals of this paper. Our goal is to suggest that SE tasks that use propositional theorem provers can benefit from phenomena reported in the AI literature; i.e. the backdoor effect. As evidence of that benefit, we point to (1) the brevity of SNAP's tests; (2) the speed with which they can be generated and fully verified, and (3) their similar diversity.

6 THREATS TO VALIDITY

One threat to the validity of this work is the *baseline bias*. Indeed, there are many other sampling techniques, or solvers, that SNAP might be compared to. However, our goal here was to compare SNAP to a recent state-of-the-art result from ICSE'18. In further work, we will compare SNAP to other methods.

A second threat to validity is *internal bias* that raises from the stochastic nature of sampling techniques. SNAP requires many random operations. To mitigate the threats, we repeated the experiments for 30 times and reported the medium or IQR of those results.

A third threat is the *measurement bias*. To determine the diversity of a test suite, in the experiment, we use normalized compression distance (NCD). Prior research has argued for the value of that measure [18]. However, there exist many other diversity measurements for the theorem proving problem, such as [10], and changing the diversity measurement might lead to a change of the results. That said, in one research report, it is impossible to explore all options.

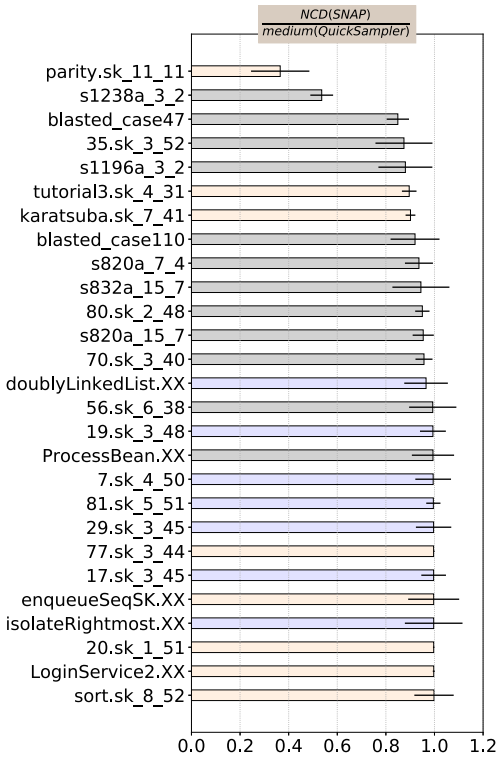

Figure 7: RQ4 results: Normalized compression distance (NCD) f or when *QuickSampler* and SNAP terminated on the same case studies. Median results over 30 runs (and small black lines show the 75th-25th variations). Same color scheme as Table 2.

For the convenient of further exploration, we have released the source code of SNAP in the hope that other researchers will assist us by evaluating SNAP on a broader range of measures.

Another threat is *hyperparameter bias*. The hyperparameter is the set of configurations for the algorithm. There now exists a range of mature hyperparameter optimizers [21, 37, 43] which might be useful for finding better settings for SNAP. This is a clear direction for future work.

Finally, as to *construct validity*, this paper argued for the benefit of backdoors by analyzing the difference between two algorithms: SNAP and *QuickSampler*. For that purpose, we used *QuickSampler* exactly as it was described in its ICSE'18 paper. Note that a case could be made to "tinker" with *QuickSampler* in order to, say, use a different termination condition. We did not do that since there are many ways we could tinker with *QuickSampler* using different parts of Figure 2. For example, tinkering could add delta mutation, or clustering, or our repair algorithm– at which point we would not be compared against the *QuickSampler* algorithm of ICSE'18 but some other algorithm of our own invention. For future work, we are exploring many of those "tinkerings". But for this paper, which is a baseline result commenting on the benefit of backdoor-based reasoning, our current approach is more justifiable.

7 CONCLUSION

Exploring propositional formula is a core computational process with many areas of application. Here, we explore the use of such formula for test suite generation. SAT solvers are a promising technology for finding settings that satisfy propositional formula. The current generation of SAT solvers is challenged by the size of the formula seen in the recent SE testing literature.

One tactic for taming the computational complexity of SAT solving is the backdoor effect. AI researchers report that when propositional formula share many settings, then fast solutions can be generated by first setting just a few of those variables. The goal of this paper was to test the efficacy of such backdoor-based reasoning. To the best of our knowledge, this is the first paper to find, and successfully exploit, backdoor effects in SE testing.

The Snap algorithm was an experiment to apply backdoors to software testing. We reasoned that the common settings seen in a small M sample of valid tests may also be the common settings in a large $N \gg M$ sample. If so, then test case generation could be made faster by sampling around the average values seen in a few randomly selected valid tests.

Experiments with Snap are strongly supportive of the benefits of backdoors for SE tasks. On experimentation, we found that when this tactic was applied to 27 real-world test case studies, Snap ran 10 to 3000 times faster (median to max) than a prior report (reported at ICSE'18). While that prior work found tests that were 70% valid, SNAP's generated 100% valid tests. Another important result was the size of the test set generated via backdoor-reasoning. There is an economic imperative to run fewer tests when companies have to pay money to run each test, and when developers have to spend time studying the failed test. In that context, it is interesting to note that SNAP's tests are 10 to 750 times smaller (median to max) than those from prior work.

In future work, we want to see if backdoors help other SE tasks that use propositional formula. For example, all the formula here have yes, no answers. Another kind of task are *optimizers* that explore satisficing trade-offs between competing constraints. To that end, we are currently working on applying backdoors to the vZ [6] optimizing theorem prover.

REFERENCES

[1] Sheldon B. Akers. 1978. Binary decision diagrams. *IEEE Trans. Computers* 6 (1978), 509–516.

[2] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. 2012. The community structure of SAT formulas. In *TAST*. Springer, 410–423.

[3] Franco Arito, Francisco Chicano, and Enrique Alba. 2012. On the application of SAT solvers to the test suite minimization problem. In *SSBSE*. Springer, 45–59.

[4] Gilles Audemard and Laurent Simon. 2009. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono DâĂŹelia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 50.

[6] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *v*Z-an optimizing SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 194–199.

[7] Sergey Bochkanov and Vladimir Bystritsky. 2013. Alglib. *Available from: www. alglib. net* 59 (2013).

[8] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. 2008. The mathsat 4 smt solver. In *International Conference on Computer Aided Verification*. Springer, 299–303.

[9] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. 2015. On parallel scalable uniform SAT witness generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 304–319.

[10] Sourav Chakraborty and Kuldeep S. Meel. 2019. On testing of Uniform Samplers. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*.

[11] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2014. Balancing scalability and uniformity in SAT witness generator. In *Proceedings of the 51st Annual Design Automation Conference*. ACM, 1–6.

[12] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 144–155.

[13] Martin Davis and Hilary Putnam. 1960. A computing procedure for quantification theory. *Journal of the ACM (JACM)* 7, 3 (1960), 201–215.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient sampling of SAT solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 549–559.

[16] Niklas Eén and Niklas Sorensson. 2006. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2 (2006), 1–26.

[17] Stefano Ermon, Carla P Gomes, and Bart Selman. 2012. Uniform solution sampling using a constraint solver as an oracle. *arXiv preprint arXiv:1210.4861* (2012).

[18] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 223–233.

[19] Johannes K Fichte and Stefan Szeider. 2015. Backdoors to normality for disjunctive logic programs. *ACM Transactions on Computational Logic* 17, 1 (2015), 7.

[20] Martin Finke. 2015. Equisatisfiable SAT Encodings of Arithmetical Operations. *http://www. martin-finke. de/documents/Masterarbeit_bitblast_ Finke. pdf* (2015).

[21] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 72–83.

[22] Serge Gaspers and Stefan Szeider. 2012. Backdoors to satisfaction. In *The Multivariate Algorithmic Revolution and Beyond*. Springer, 287–317.

[23] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2013. Advanced conflict-driven disjunctive answer set solving. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

[24] Vibhav Gogate and Rina Dechter. 2011. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence* 175, 2 (2011), 694–729.

[25] Marijn Heule, Matti Järvisalo, and Tomas Balyo. 2017. Sat competition. *SAT* (2017).

[26] Mahesh A Iyer. 2003. RACE: A word-level ATPG-based constraints solver system for smart random simulation. In *null*. IEEE, 299.

[27] Mikoláš Janota, Victoria Kuzina, and Andrzej Wąsowski. 2008. Model construction with external constraints: An interactive journey from semantics to syntax. In *MoDELS*. Springer, 431–445.

[28] 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*. ACM, 177–187.

[29] Martin Kronegger, Sebastian Ordyniak, and Andreas Pfandler. 2014. Backdoors to planning. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*.

[30] Joohyung Lee and Vladimir Lifschitz. 2003. Loop formulas for disjunctive logic programs. In *ICLP*. 451–465.

[31] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. 2009. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM (JACM)* 56, 4 (2009), 22.

[32] Ming Li and Paul Vitányi. 2013. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media.

[33] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. 1993. The computational complexity of universal hashing. *Theoretical Computer Science* 107, 1 (1993), 121–133.

[34] Kuldeep S Meel, Moshe Y Vardi, Supratik Chakraborty, Daniel J Fremont, Sanjit A Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. 2016. Constrained sampling and counting: Universal hashing meets SAT solving. In *Workshops at the thirtieth AAAI conference on artificial intelligence*.

[35] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, 231–240.

[36] Tim Menzies, David Owen, and Julian Richardson. 2007. The strangest thing about software. *Computer* 40, 1 (2007), 54–60.

[37] Vivek Nair, Amritanshu Agrawal, Jianfeng Chen, Wei Fu, George Mathew, Tim Menzies, Leandro Minku, Markus Wagner, and Zhe Yu. 2018. Data-driven search-based software engineering. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 341–352.

[38] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 11.

[39] Bart Selman, Henry A Kautz, Bram Cohen, and others. 1993. Local search strategies for satisfiability testing. *Cliques, coloring, and satisfiability* 26 (1993), 521–532.

[40] Willem Visser, Corina S Pasareanu, and Radek Pelánek. 2006. Test input generation for java containers using state matching. In *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 37–48.

[41] Wei Wei, Jordan Erenrich, and Bart Selman. 2004. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, Vol. 4. 670–676.

[42] Ryan Williams, Carla P Gomes, and Bart Selman. 2003. Backdoors to typical case complexity. In *IJCAI*, Vol. 3. 1173–1178.

[43] Tianpei Xia, Rahul Krishna, Jianfeng Chen, George Mathew, Xipeng Shen, and Tim Menzies. 2018. Hyperparameter Optimization for Effort Estimation. *arXiv preprint arXiv:1805.00336* (2018).

[44] Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, Yutaka Oiwa, and Armin Biere. 2015. Optimization of combinatorial testing by incremental SAT solving. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[45] Zhe Yu, Fahmid M. Fahid, Tim Menzies, Gregg Rothermel, Kyle Patrick, and Snehit Cherian. 2019. TERMINATOR: Better Automated UI Test Case Prioritization. In *FSE'19 (SEIP)*.

[46] Jun Yuan, Adnan Aziz, Carl Pixley, and Ken Albin. 2004. Simplifying Boolean constraint solving for random simulation-vector generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 3 (2004), 412–420.

[47] Jun Yuan, Kurt Shultz, Carl Pixley, Hillel Miller, and Adnan Aziz. 1999. Modeling design constraints and biasing in simulation using BDDs. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 584–590.