Search-based Fuzzing

László Szekeres
Google

R. Sekar
Stony Brook University

ABSTRACT

Fuzzing and dynamic symbolic execution (DSE) are the two main approaches currently being used for automated bug finding. The key strength of fuzzing is that it is simple to use, and can generate and evaluate a very large number of test cases in a short time. Its main drawback is its lack of direction (or blindness), which means that the vast majority of test cases are not useful, i.e., they do not identify new bugs or even contribute to increasing test coverage. It is in this regard that symbolic execution excels: by employing constraint solvers, DSE techniques can generate inputs that can target specific branches of conditions, thereby providing a reliable way to increase coverage. However, targeted input generation is far slower than the test generation techniques used in fuzzers, leading to far fewer test cases being explored every second. We propose a new approach in this paper called *search-based fuzzing (SBF)* that spans the gap between the two extremes represented by fuzzing and DSE. SBF's input generation is much more targeted than today's fuzzers, yet significantly faster than symbolic execution techniques. This combination seems very promising: our evaluation shows that SBF can achieve significantly more coverage than state-of-the-art tools for fuzzing and symbolic execution.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*;

KEYWORDS

software security, testing, fuzzing, automated test generation

1 INTRODUCTION

There are two main automated testing approaches for finding memory corruption bugs: fuzzing and dynamic symbolic execution (DSE). Fuzzing [31] involves running the program under test with randomly mutated inputs and detecting crashes. It is simple to use, requiring little user effort. Fuzzing introduces very little (runtime or memory) overhead to the test runs, so it scales well. These factors make fuzzing a popular choice among security practitioners. Many of the infamous vulnerabilities of the past few years, such as Heartbleed [10], Shellshock [44], and Stagefright [13], were found with fuzzing.

Despite these successes, fuzzing tools are limited in their ability to cover large parts of the code. Simple conditions, such as the one shown in the code snippet below, may require billions of test cases to be generated before arriving at one that satisfies the condition. This is because fuzzers generally mutate inputs *blindly*, without considering program logic or structure.

```
void fuzzme(int input) {
  if (input == 0xbadc0de) { /* some vulnerable code */ }
}
```

In contrast, DSE [8] generates inputs that target specific conditions to "punch" through. Programs are run with symbolic inputs in order to generate a formula that captures the constraints necessary to take a particular program path. These constraints are then solved using a constraint solver to obtain a concrete input that traverses the targeted path. Using this approach, DSE systems can generate inputs that get past relatively complex conditions.

The systematic approach embedded in DSE systems has the potential to exercise most program paths, and thus achieve good code coverage. In practice, however, this potential is limited by the costs and complexities associated with DSE. As a result, fuzzing tools can often outperform DSE in practice. In their recent SoK paper, Shoshitaishvili et al. [39] reported that fuzzing tools identified almost three times as many vulnerabilities as DSE techniques.

Trying to harness the best of fuzzing and DSE, researchers have developed hybrid systems that first employ fuzzing to generate test inputs, with DSE techniques applied whenever the fuzzer stops making progress. During the recent DARPA Cyber Grand Challenge competition most of the teams, including the top three, used this hybrid approach [3, 19, 30, 40]. Although such a combination is effective in avoiding the worst outcomes of the two techniques, it does not fundamentally address the underlying problems of either approach: the fuzzer component continues to operate blindly, while the DSE component continues to be challenged by large and complex code bases.

In contrast, we present a new approach called *search-based fuzzing (SBF)* that represents an intermediate point between the two extremes of blind and highly-targeted input generation. SBF uses two systematic techniques, namely, data-flow tracking and stochastic local search to bring considerable directionality to fuzzing. Although not nearly as precise as DSE in its targeting, SBF is able to avoid many of the complexities of DSE, and hence can more easily scale to large and complex programs.

1.1 Approach Overview

Automated test generation tools make a trade-off between:

(1) the computation needed to generate a new input, and
(2) the likelihood of increased coverage due to this input.

Fuzzing, at one end of the spectrum, spends very little time on input generation, but these randomly generated inputs have a very low probability of increasing coverage. At the opposite end of the spectrum, DSE generates inputs with a high probability of increasing coverage, but to do so, it expends a lot of resources in generating each input. We present a new approach called *search-based fuzzing* (SBF) that falls in between, thus representing a more favorable trade-off between the two extremes.

Like DSE, SBF is *directed:* it targets specific conditional branches in order to increase coverage. Like fuzzers, it needs only a lightweight source instrumentation, thereby avoiding the complexity of DSE, and the performance costs of constraint solvers.

Unlike fuzzers that tend to operate blindly, SBF targets its input mutations in order to reach a specific branch condition. It uses *data-flow tracking* (DFT) to determine ***what*** parts of the input affect the targeted branch condition, and targets its mutations to those bytes. It then uses a *stochastic local search* (SLS) [20] to determine ***how*** to mutate these bytes in order to take the targeted branch. Both DFT and SLS are achieved using a light-weight instrumentation, thereby enabling SBF to retain the performance benefits of fuzzing.

By combining some of the key strengths of DSE and fuzzing, namely, directionality and performance, SBF can be more effective than either of these approaches. We provide a few examples here to illustrate this strength:

- *"Magic value" example:* A blind fuzzer needs about 2 billion attempts (on average) before it generates the input `0xbadc0de` that passes the condition. In contrast, SBF requires just about 32 attempts using SLS.
- *Adler checksum:* Used in popular software packages such as `zlib` and `rsync`, this checksum function is considerably more complex than the simple comparison above. Given a condition such as `adler(s)==0xbadc0de`, our experience with fuzzers such as AFL is that they need several hours, if not more, to get through the condition. In contrast, SBF completes this task in seconds using its combination of SLS and DFT.
- *Maze example:* Finally, consider the maze example from the KLEE tutorial [25]. Solving the maze requires successfully passing over two dozen branches, a tall order for most fuzzing tools. SBF is able to solve the maze in a fraction of a second.

These examples, together with our experimental evaluation results, demonstrate that SBF is able to achieve more coverage in lesser time than state-of-art fuzzing or DSE tools. In particular, SBF:

- reaches 2× higher coverage in a few minutes than that achieved by other tools after running for a few hours.
- finds all the bugs in a LAVA modified benchmark program in under a minute, while other tools find only a subset of the bugs after five hours.

Contributions. We introduce *search-based fuzzing*, a new automated test generation technique, based on stochastic local search. It has the following three main components:

- *search target identification*, used to select the parts of the program to target, and identify *which* bytes of the input need to be mutated to reach those parts;
- *local search based mutation strategy,* which indicates *how* to mutate these bytes in order to reach the selected target program point. We propose a special *stochastic local search* based on the Markov Chain Monte Carlo (MCMC) algorithm.
- *test suite inflation and deflation* technique, which overcomes the limitations of edge coverage metric while containing path explosion.

Another contribution of the paper is the open-source SBF tool, set to be released together with the paper.

2 SEARCH-BASED FUZZING DESIGN

To provide feedback for input generation, we instrument the comparison instructions in the target program. We refer to comparison instructions as *nodes* and their two outcomes as *edges*. Note that complex if-statements will get translated into multiple nodes. We represent our control-flow graphs (CFG) in terms of these nodes and edges, as shown in Figure 1 for the following example:

Listing 1 Printable string example

```
bool str_isprint(unsigned char *data) {
  for (; *data != '\0'; data++)
    if (!(0x1f < *data && *data < 0x7f))
      return false;
  return true;
}
```

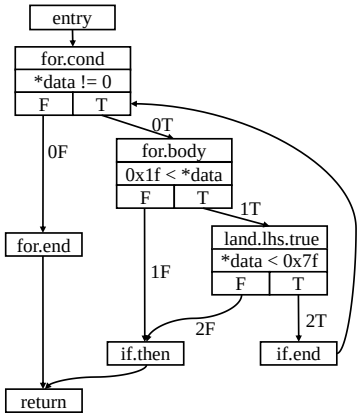

Figure 1: CFG of the "printable string" example.

We illustrate our approach on this example using an initial input, a single-byte vector `[0x0]`. This input takes the 0F path in Figure 1. Running this input through our target identification algorithm, the algorithm identifies that the true edge of node 0 (0T) has never been taken before and that it depends on the first input byte. Because of this, the edge 0T, together with its dependencies, is flagged as a local search target. A stochastic local search is carried out to find a value assignment to the relevant bytes that makes the execution take the targeted edge. This means that we associate a distance function with the targeted edge. Using the distance function as a feedback, the local search will quickly modify the first byte to, say, `0x1`, which will exercise the path [0T, 1F].

Having observed new coverage of the edge 1F, we next proceed to target the edge 1T with a local search. The distance value for an input assignment can be based on the Hamming-distance between the two operands of the targeted compare node. With the input `[0x1]`, the targeted comparison is `0x1f < 0x1` and the Hamming distance is 4. Reducing this distance gets us closer to negating the outcome of the comparison. When the search algorithm modifies the affecting byte, it uses this distance as a feedback to decide if a certain mutation (e.g., a bit flip) is useful or not. Through iterative improvements, the algorithm quickly finds an input, say `[0x2f]`, for which the distance is 2 and the execution takes the [0T, 1T, 2T] path. The test generation continues in this manner by targeting

further uncovered edges, or even increasing the loop counts by targeting taken edges more times than before.

This high level strategy resembles that of dynamic symbolic execution systems. Say our initial input exercises the path with the condition $(data[0] \neq 0) \wedge (\texttt{0x1f} < data[0]) \wedge (data[0] < \texttt{0x7f})$. In order to exercise a different path, we want to negate the outcome of a node, say, $(data[0] < \texttt{0x7f})$. This yields the new path condition $(data[0] \neq 0) \wedge (\texttt{0x1f} < data[0]) \wedge \neg(data[0] < \texttt{0x7f})$ that is targeted next. In contrast with DSE, our algorithm does not know about this formula. It only uses the distance value which is computed by the instrumented program.

Another significant difference with symbolic execution is that we try to find a satisfying input by focusing on just the last condition. This assumes that mutating the bytes that influence the targeted edge will not invalidate earlier branch conditions. This is an overly optimistic assumption, and as a result, only a subset of new inputs generated by SBF will follow the targeted program path. At the same time, our assumption enables new test cases to be generated incrementally with each additional branch condition, something that is generally difficult in symbolic execution. The result is an input generation step that can be orders of magnitude faster, making up for lack of precision by allowing SBF to run many more iterations of its search within the same time. Our experimental results suggest that this speed makes up for the increased error in targeting that may result due to a violation of our assumption.

2.1 Coverage Map and Distance Map

We instrument all nodes (compare instructions) in the program to compute two mappings on edges (potential outcomes): the *coverage map* C and the *distance map* $\mathcal{D}$. For an edge e, $C[e]$ is the number of times the edge was passed. For each comparison c that was executed, $\mathcal{D}$ specifies a distance value for its last execution. For a taken edge e_{taken}, $\mathcal{D}[e_{taken}] = 0$; for a non-taken edge $e_{\neg taken}$, $\mathcal{D}[e_{\neg taken}] = d(c)$, where c is the comparison preceding the edge. The distance function $d\colon C \mapsto (0, 1]$, maps a comparison $c = x^{[n]} \bowtie y^{[n]}$ to a non-zero value, where n denotes the bit width of the comparison. We implemented two distance functions: Hamming distance $d_H(c)$, given by Equation (1), and arithmetic distance $d_D(c)$, given by Equation (2). Note that in these equations, $+/-$ is defined with overflow semantics.

$$d_H(c) = \begin{cases} 1/n \cdot max(1, H(x, y)) & \text{if } \bowtie \in \{=, \neq, \leq, \geq\} \\ 1/n \cdot max(1, H(x, y) + 1) & \text{if } \bowtie \in \{<, >\} \end{cases} \quad (1)$$

$$d_D(c) = \begin{cases} 2^{-n} \cdot max(1, |x - y|) & \text{if } \bowtie \in \{=, \neq, \leq, \geq\} \\ 2^{-n} \cdot max(1, |x - y| + 1) & \text{if } \bowtie \in \{<, >\} \end{cases} \quad (2)$$

The distance function serves as our score function for local search. It represents how close we are from taking a non-taken edge, so it is always larger than 0. We call the non-taken edges *touched edges*, because the execution traversed the corresponding compare instruction, but the other edge was taken. Edges belonging to non-executed nodes are assigned the maximum distance value.

2.2 Main Fuzzing Cycle

Algorithm 1 shows our high-level search algorithm. It has similarities to coverage-guided fuzzing [6, 37, 43], as well as DSE systems. This main function takes an existing set of test cases. If this set is empty, then we generate a seed test case consisting of all zeroes. The algorithm generates new test cases from existing ones. Test cases are processed in a work list (queue). Each test case in the list is fuzzed, and among the resulting mutants, those that achieve new coverage are added back to the list. The algorithm returns when the work list is exhausted. This is what we call one fuzzing cycle.

Algorithm 1 Main Search-based Fuzzing Cycle

Input: initial test suite *TestSuite*
Output: new test suite *TestSuite* and bug triggering inputs *Crashers*

```
1:  function SbfCycle(TestSuite)
2:      if TestSuite = ∅ then
3:          TestSuite ← {"00. . . 0"}
4:      WorkList ← TestSuite
5:      TestSuite ← ∅
6:      while WorkList ≠ ∅ do
7:          t ← PopOneFrom(WorkList)
8:          T ← IdentifySearchTargets(t)
9:          for each target ∈ T do
10:             LocalSearch(t, target)
11:         for each t' ∈ ByteMutations(t) do
12:             RunAndCheck(t')
13:         TestSuite ← TestSuite ∪ {t}
14:     return TestSuite
15: function RunAndCheck(t')
16:     C_run, D ← Execute(t')
17:     if CrashHappened() then
18:         Crashers ← Crashers ∪ {t'}
19:     if C_run ⊄ C_global then
20:         C_global ← C_global ∪ C_run
21:         WorkList ← WorkList ∪ {t'}
22:     return D
```

To find new paths, SBF uses a more systematic, targeted strategy than existing blind coverage-guided fuzzers. For each fuzzed input SBF first establishes the set of potentially reachable new edges using a *target identification phase*, implemented by IdentifySearchTargets(); and then tries to reach them by carrying out a directed *local search* for each of those targets. The vast majority of the fuzzing time is spent in this local search phase (LocalSearch()). Optionally, an additional *blind fuzzing* step is run after this phase (the second for loop), which we will discuss more later.

The *target identification phase* identifies the edges towards which we direct our local searches. Our goal here is to negate the outcome of the nodes (comparisons) along the path taken by the initial input so as to cover previously uncovered edges. One main difference between blind fuzzing and search-based fuzzing is that we first identify these *touched* edges and direct our input mutations towards them. Trying to flip the taken branches is similar to the generational search strategy [18] of concolic execution systems.

The output of the target identification phase is the list of control dependent touched edges that if reached, would increase the global coverage. With each edge, we provide the list of input byte indices that affects the given edge. Input dependent means that the (controllable) input of the program (i.e., the attack surface) has an effect

on that particular comparison (or branch). We use byte-precise data-flow tracking (Section 2.3) to identify input dependencies. Once the targets are identified, a local search algorithm (LocalSearch()) is run for each target. This local search aims to take the target edge by changing the affecting input bytes, using the distance map as feedback. We discuss this phase in Section 2.4.

Finally, in the optional *blind fuzzing* step, random blind mutations are done on each byte of the test vector, similarly to the random bit and byte flips of existing (black or gray-box) fuzzers. The ByteMutations() function returns a set of mutations of the input, where only a single byte is modified in each mutant. This phase is optional, as its primary purpose is not to find new paths, but to trigger bugs along the original path triggered by the input. For instance, considering the single path program `char buf[10]; buf[input]=0;`, we might cover the path with input 5, but the bug is only triggered with input outside of the $[0, 9]$ range.

With each mutation carried out during a local search or during the blind fuzzing phase, the program is run with the RunAndCheck() function, shown in Algorithm 1. Each time the coverage map is generated to detect new coverage and the distance map is generated to serve as a feedback for the local search.

2.3 Search Target Identification

The goal of the target identification phase is twofold: (1) to identify the set of nodes (and their edges) influenced by input I and (2) for each of those nodes, establish the set of input byte locations that affect the outcome of the given node. We will target the touched edges that the current input has an effect on and could increase the coverage (as defined in Section 2.5). We will focus each targeted local search to the input bytes that affect the targeted edge, thus reducing the search space of the local search to focus only on the bytes that matter.

The target identification algorithm uses byte-precise data-flow tracking. We assign different labels to each input byte and we propagate the labels of all memory locations and registers during execution. Our instrumentation at each comparison checks if the arguments depend on the input, and if they do, on which input bytes exactly. After a single execution with data-flow tracking, the analysis returns a set of search targets. Each target is a pair, consisting of a yet uncovered reachable (touched) edge, and the corresponding set of input dependencies.

2.4 Stochastic Local Search

Once the search targets are established by the target identification phase, we start a stochastic local search for each target. During the local search we mutate the bytes that affect the target branch, with the guidance of a distance function. For instance, the distance function for the edge corresponding to the condition (`100 < sum`) in Listing 2 looks like Figure 2. (We are using the distance function given by Equation (2)).

Listing 2 Simple example

```
void sum(byte input[2]) {
  byte x = input[0], y = input[1], sum = x + y;
  if (100 < sum && sum < 150) { /* some vulnerable code */ }
}
```

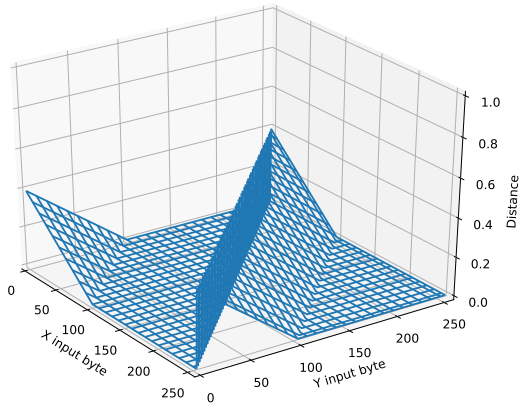

Figure 2: Distance function for edge $(100 < x + y)$**.**

The search algorithm can use this distance information as a feedback to determine if a potential mutation on the input would get us closer to the solution or not. A mutation can be considered as a move from one point to another on the plot. By moving more intelligently on this distance landscape than randomly, we can find a satisfying solution point faster. Next we discuss the exact algorithms we use.

2.4.1 Local search algorithms. A local search algorithm is composed of the following three components:

(1) a distance function (also called the score function),
(2) a definition of neighbors,
(3) and a search strategy.

The distance function assigns a value to a point in the search space representing how *far* it is from a potential solution. The definition of neighbors determines what points do we consider *neighboring points* in the search space, the ones reachable in a single step. Finally, the search strategy defines how the search moves from point to (neighboring) point, using the distance function as a guide.

In SBF we implemented two different distance functions, two neighbor definitions and six different search strategies. We evaluated all possible combinations of these on a variety of benchmarks. We found that different algorithms work best for different targets, yet this evaluation helped us design an algorithm that works well on a wide spectrum of targets. In this section we describe this main algorithm along with the others we tested. The SBF tool uses this main local search algorithm by default, and the experiments in the evaluation section were carried out using it as well.

The two distance functions in SBF are the ones defined in Equation (1) and (2). The two neighbor functions we considered are *BitFlip* and *AddSub*. *BitFlip* defines neighbors to be those that differ in exactly one bit position. *AddSub* adds (or subtracts) powers of two to get to a neighbor. The neighbors of an input vector only differ in the bytes that actually affect the targeted edge. In other words, all mutations are done solely on the bytes that the targeted edge depends on.

Next we describe the search strategies we implemented and evaluated. Our first algorithm, *Random Walk* shown in Algorithm 2 makes a random move in every step. It is not a *real* local search

Algorithm 2 Random Walk

Input: Initial test vector I, target edge e, and the list of byte indices $l = (i_0, i_n, \ldots, i_k)$ that e depends on.

Output: Test vector reaching e, or nothing. (Note, that implicitly RunAndCheck() also saves all input that triggers new coverage.)

```
1: for i ← 1 to max_steps do
2:     I ← PickRandomly(Neighbors(I, l))
3:     D_I ← RunAndCheck(I)
4:     if D_I[e] = 0 then
5:         return I
```

Algorithm 3 Hill Climbing with Eagerness Probability Extension

Input: Initial test vector I, target edge e, list of byte indices $l = (i_0, i_n, \ldots, i_k)$ that e depends on, and the probability of taking non-worsening move eagerly (without evaluating all neighbors) p_{eager}.

Output: Test vector reaching e, or nothing. (Note, that implicitly RunAndCheck() also saves all input that triggers new coverage.)

```
1: for i ← 1 to max_steps do
2:     progress ← True
3:     while progress do
4:         progress ← False
5:         N_best ← I
6:         for n ← 1 to number_of_neighbors do
7:             N ← Neighbor_n(I, l)
8:             D_N ← RunAndCheck(N)
9:             if D_N[e] = 0 then
10:                return N                    ▷ Found a solution.
11:            if D_N[e] < D_{N_best}[e] then
12:                progress ← True
13:                N_best ← N
14:                if Random([0, 1)) < p_eager then
15:                    I ← N                   ▷ Make move eagerly.
16:        I ← N_best                          ▷ Make the best move.
17:    I ← PickRandomly(Neighbors(I, l))
```

algorithm, as it does not use the distance function for guidance, only to decide whether we have found a solution so the search can stop. We will only use this algorithm as a baseline in our experiments. In the following algorithms $\mathcal{D}_I[e]$ denotes the distance of an input I from edge e. $Neighbor_n(I, l)$ is the n-th neighbor of I, where l is the list of affecting byte indices.

Perhaps the most well-known local search algorithm is the *Hill Climbing* algorithm. It is a greedy algorithm, which in each step evaluates all neighbors of the current point and moves to the one with the best distance. Doing this we can get stuck in local minima, for which the common solution is to make random moves until we can make progress again. We also follow this approach. We extend the basic hill climbing with an *eagerness probability*. This addition is shown on lines 14 and 15 of Algorithm 3. During the evaluation of the possible moves, if the currently evaluated neighbor improves the distance, with some probability we take that move before evaluating all possible moves. When we eagerly move when evaluating the ith neighbor, we continue by evaluating the $i + 1$th neighbor of the new point.

In its limit, when the eagerness probability is 1, the algorithm evaluates the possible moves in order and immediately takes any improving one. This is especially useful when we target an equality, e.g., `if (input == magic_number)`. Suppose we use the *BitFlip* neighbor definition, and Hamming distance as the distance function. In this case, the local search algorithm will go through each bit of `input` one by one, and flip them if that particular bit is different in `magic_number`. This algorithm is guaranteed to find the right input in maximum as many steps as the number of bits in `input`. When eagerness probability is one we will refer this algorithm as *Eager*, otherwise we call it Hill Climbing.

Our next algorithm is Simulated Annealing and its special case, Markov Chain Monte Carlo (MCMC) sampling. This strategy does not evaluate all options in every step, but picks a random neighbor, then decide whether to move there or not based on the neighbor's distance value. When the distance is better (smaller), we always make the move. However, to be able to escape from local minima, we also accept non-improving moves with some probability. This probability is determined by the Metropolis-Hastings acceptance criteria, shown on line 11 of Algorithm 4.

Originally, MCMC is a method for sampling from probability distributions, so that we take samples from regions with higher probability density more often than regions with lower probability. However, the technique can be directly applied as a local search algorithm, sampling from a search space according to a distance function. This type of stochastic local search, has been used by others as well, for instance, by STOKE [35] for superoptimization.

Simulated Annealing is another adaptation of the Metropolis-Hastings acceptance criteria, where additionally to the MCMC algorithm we also lower the acceptance probability with time. It adds a T *temperature* factor to the algorithm, which is reduced according to a γ cooling factor, as shown on line 13. In the extreme case when $\gamma = 1$, the two versions of the algorithm is equivalent. In this special case, we will refer to this algorithm as *MCMC*, otherwise as *Simulated Annealing*.

Our final algorithm, which we designed particularly for SBF, is the combination of the Eager and MCMC algorithms. We run the the above described Eager (Algorithm 3 with $p_{eagerness} = 1$) until that strategy stops making progress (the first time we get stuck), then we switch to MCMC. As the following benchmarks confirm, this helps in finding solutions for easy targets, such as magic values as quickly as possible, while also allows solving hard targets. We refer to this algorithm as *EagerMCMC*.

In order to evaluate the described algorithms, we ran all possible combinations of the two distance functions, two neighbors definitions, and six search strategies, with different parameters, on a set of benchmarks. Considering the different parameters (e.g., for β or $p_{eagerness}$), this adds up to 236 different configurations of local search algorithms. We tried these configurations on ten small benchmarks. Each benchmark does some computation on the input and compare the result of that computation to some constant (i.e., equals/larger than). The computations include calculating different checksums, such as modular sum, Adler, and Fletcher. Others

Algorithm 4 MCMC and Simulated Annealing

Input: I test vector, e target edge, $l = (i_0, i_n, \ldots, i_k)$ list of byte indices e depends on, β accept probability factor, γ cooling factor

Output: Test vector reaching e, or nothing. (Note, that implicitly RunAndCheck() also saves all input that triggers new coverage.)

$T_0 \leftarrow 1.0$
for $i \leftarrow 1$ **to** *max_steps* **do**
  $N \leftarrow$ PickRandomly(Neighbors(I, l))
  $\mathcal{D}_N \leftarrow$ RunAndCheck(N)
  if $\mathcal{D}_N[e] = 0$ **then**
    return N ▷ Found a solution.
  $\Delta \leftarrow \mathcal{D}_N[e] - \mathcal{D}_I[e]$)
  if $\Delta < 0$ **then**
    $I \leftarrow N$ ▷ Improvement found.
  else
    if Random($[0, 1)$) $< e^{\frac{-\Delta}{\beta \cdot T_i}}$ **then**
      $I \leftarrow N$ ▷ Acceptance criteria passed.
  $T_{i+1} \leftarrow \gamma \cdot T_i$

convert the input string to an integer or a float number, or implement a polynomial function and some other simple functions. The goal of each search in the benchmark is to find an input that takes to true edge of the comparison.

We ran all configurations on all benchmarks 1000 times, with *max_steps* = 100000, and measured how many times the search found a solution. Table 1 summarizes the results. We do not list all 236 configurations, only the best configuration for the six different algorithms we described earlier. We indicate the distance and neighbors function used in the configuration, and the parameters that were used for the search strategy.

The combined EagerMCMC algorithm, using Hamming distance and AddSub neighbors definition had the highest success rate. MCMC combined with the Eager algorithm does significantly better only on the easiest targets (e.g., input equals to a constant), and similarly on the other ones, therefore the overall success rate is just marginally better. We also found that Simulated Annealing results were always worse than MCMC. This is why Simulated Annealing

Search strategy	Distance	Neighbors	Success rate
EagerMCMC ($\beta = 0.2$)	Hamming	AddSub	94.81%
MCMC ($\beta = 0.2$)	Hamming	AddSub	94.54%
SimulatedAnnealing ($\beta = 0.2, \gamma = 0.999$)	Hamming	AddSub	92.89%
HillClimbing ($p_{eagerness} = 0.1$)	Difference	AddSub	89.08%
Eager	Difference	AddSub	80.23%
RandomWalk	Difference	BitFlip	10.27%

Table 1: The best configuration of each algorithm and its average success rate on our benchmark.

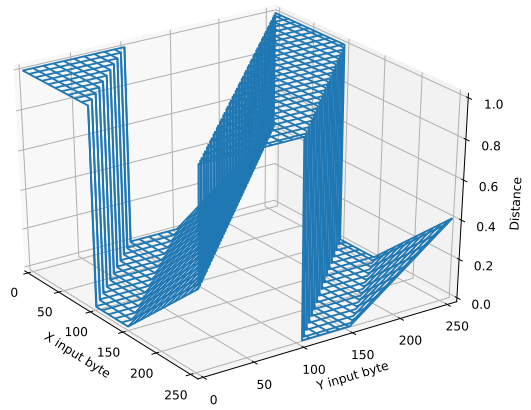

Figure 3: Distance function for edge $(x + y < 150)$**.**

performed the best when its γ parameter was closes to MCMC's. Hill Climbing did worse than MCMC in general, but interestingly the best performing configuration was when the eagerness probability was 0.1. This means that our eagerness extension improves on the standard algorithmm, where this probability is zero (or unused).

2.4.2 Targeting paths vs. targeting edges. There are two ways we can direct a search towards an uncovered edge. We can target a specific path prefix leading to the edge we want to reach, or we can disregard the path prefix and only focus on the comparison whose outcome we want to negate. We call the first *path targeting*, and the other *edge targeting*. In case of path targeting, the distance feedback for the local search needs to take into account all nodes along the path prefix. We can define such *path distance* as the sum of the node distances along the targeted path: $\sum_{\forall e \in Path} \mathcal{D}[e]$. For edge targeting, we simply use $\mathcal{D}[e]$ as the feedback.

Recall that we only target edges that are *touched* by the path of the currently mutated input. This means that if we target an edge, the current input already exercises the path prefix leading to it, so all node distances along that path prefix will be zero. In other words, initially the path distance and the edge distance will be the same. We rely on this fact to make an optimization and only take into account the distance for the targeted edge, not considering its path prefix, and the dependencies of all prior nodes. This, however, is an opportunistic optimization, because when we start mutating the input, we can easily get off the original path prefix and no longer reach the targeted node.

As an example, consider the distance function in Figure 3, which represents the distance for edge corresponding to the condition `sum < 150` in Listing 2. This node is guarded by the condition `100 < sum`. If this condition is not satisfied, we do not reach the second node. These are the points with non-zero distance in Figure 2 on page 4. Recall, that for unreached edges we assign maximum distance, which creates a plateau in the distance landscape, which hinders guiding the local search. However, plateaus only cause a problem if we start the search from them. In our example we are guaranteed to start a search from a point outside the plateau areas, because we target this edge from an input that already passes the first comparison. In other words we start from one of the zero-distance points in Figure 2, satisfying the $100 < x + y$ condition.

Our targeting of edges as opposed to specific paths reduces the search space significantly and works well when the preceding conditions along the path prefix are independent from the target condition. The above example also shows that it can work well when consecutive conditions are on the same input bytes. Another benefit of directing the search towards an edge, and not forcing it to take a specific path, is that it provides more opportunities for expanding coverage beyond the target using cheap mutations. Thus, SBF's strategy represents a middle ground between coverage-guided fuzzing, which does not target anything, and DSE, which targets a specific path.

2.5 Coverage Metric

The SBF design focuses on increasing branch coverage: it targets edges and uses edges as a coverage metric. The coverage metric determines whether an input (mutation) is new coverage, and should be added to the test suite. Specifically, we use counted edge coverage, which means that we consider an input new if (a) it covers a new edge, or (b) it covers a previously covered edge more times than before. This metric is similar to that introduced by AFL, and used by existing coverage-guided fuzzers, however there is an important difference. Recall that SBF deals with the edges of comparison nodes, while existing tools [37, 43] deal with the edges of the control-flow graph. Tools measuring CFG-edges register jumps from one basic block to another, by instrumenting jump instruction and/or basic blocks. The edge coverage metric of SBF registers the outcome of individual comparisons, by instrumenting compare instructions only.

This has multiple benefits. First of all, we only have to instrument compare instructions for maintaining both the coverage map (to target and detect new coverage), and the distance map (to guide mutation). Second, by instrumenting individual compare instructions we can effectively attack the problem of complex branch conditions. When a CFG edge is only taken if a complex branch condition is satisfied (e.g., `if (a && b && c)`), a fuzzer relying on CFG edge coverage can only detect when all sub-conditions become true. In contrast, using SBF's metric, we can detect incremental improvements, e.g., when one or two of the sub-conditions get satisfied. This more fine grained coverage feedback helps making progress, even without SBF's targeted search strategy, but also using just blind mutation.

2.6 Test Suite Inflation and Deflation

Relying on (counted) edge coverage have certain limitations. To demonstrate the issues, consider the example in Listing 3, which is taken from a KLEE tutorial [25]. The function takes a string, where each character represents taking a step in a maze. To win the game the character sequence must lead through the maze reaching the goal shown as #, i.e., the string `ddddrrrruulluurrrrddddrruuuu` will reach the "You win!" edge. Let us consider what inputs can be generated using different coverage metrics. The four interesting edges (other than the winning one) are the ones corresponding to taking the four different directions. With (uncounted) edge coverage, the test generation stops after generating the inputs `u`, `d`, `l`, and `r`, because with these we already covered taking all four directions. With *counted* edge coverage we can also find `dd`, `ddd` and `dddd`, because covering the same edge more times constitutes as new coverage. However, it will get stuck at that point and will not find `ddddr`, because it already generated `r`, in other words, we already covered "moving right once".

Listing 3 Maze example

```
int walk_maze(char *steps) {
  char maze[H][W] = { "+-+---+---+",
                      "| |     |#|",
                      "| | --+ | |",
                      "| |   | | |",
                      "| +-- | | |",
                      "|     |   |",
                      "+-----+---+" };
  int x = 1; int y = 1; // Player position.
  for (int i = 0; i < MAX_STEPS; i++) {
    switch (steps[i]) {
      case 'u': y--; break;
      case 'd': y++; break;
      case 'l': x--; break;
      case 'r': x++; break;
      default: printf("Bad step!"); return 1; }
    if (maze[y][x] == '#') {
      printf("You win!"); return 0; }
    if (maze[y][x] != ' ') {
      printf("You lose."); return 1; } }
  return 1; }
```

The problem is that in order to generate that input, we would need to take a previously covered edge, but from a different context. One potential solution for this problem would be to differentiate between all paths or path prefixes. This, however, would lead to a test suite blow up due to the practically infinite number of paths in any non-trivial program. We solve this problem differently and eliminate the limitation of counted edge coverage while avoiding using a path coverage metric.

Algorithm 5 Test Suite Inflation & Deflation

Input: initial test suite $TestSuite$
Output: new test suite $TestSuite$

1: **while** not stopped **do**
2: $InflatedTS \leftarrow$ SbfCycle($TestSuite$)
3: $DeflatedTS \leftarrow$ SetCover($InflatedTS$)
4: $TestSuite \leftarrow$ Randomize($DeflatedTS$)
5: $C_{global} \leftarrow \emptyset$ ▷ Reset global coverage map.
6: **return** $TestSuite$

We introduce an outer loop shown in Algorithm 5 around the main fuzzing cycle, SbfCycle() (Algorithm 1). Every time the work list gets exhausted, we take the generated test suite, minimize it, randomize it, and restart the main fuzzing cycle with the global coverage reset to empty. Resetting the global coverage allows subsequent cycles to rediscover the same edges, but from a different context, through the mutation of a different input. Preferably, we want to re-take edges from a deeper point of the execution than before. This is why we minimize the test suite after each cycle using a naïve approximating set covering algorithm. We eliminate redundant test cases that cover edges that are also covered by inputs

that cover more edges. In other words, we prioritize the test cases that exercise longer paths. This way the next cycle can start from a better input, and by rediscovering some edges from deeper in the program, newer states might be reached. We do not want however to always start fuzzing the input with the deepest path, therefore before each cycle we also randomize the order of the test cases. This makes sure that edges are rediscovered from a variety of contexts in each fuzzing cycle.

We call this technique *test suite inflation & deflation*, because in each round, the main fuzzing cycle inflates the test suite and the outer loop deflates it. The inflation is because, due to the cleared global coverage, a number of new inputs will be added to the test suite. Even when these are redundant from the edge coverage perspective, they exercise new paths, which enables reaching completely new edges. Typically, after the first cycle, each inflation approximately doubles the test suite size and each deflation halves it. With this strategy, we avoid test suite blow up, while we monotonically increase the global coverage.

In case of the running example, after the first cycle that gets stuck with the input `ddddr`, the second cycle will likely start by fuzzing this input, take the `r` edge from it and proceed in a similar fashion until everything is covered. Indeed, SBF reaches 100% coverage in this example in just a few microseconds.

3 IMPLEMENTATION

The SBF implementation has two parts, the SBF compiler and the SBF library. The SBF compiler extends the LLVM/Clang compiler with a plugin that performs the instrumentation. The SBF library contains the implementation of the test generation algorithms described so far. Like LibFuzzer, SBF links to the instrumented target with the SBF library in order to create a single executable. As a result, fuzzing can execute within a single process. Also like LibFuzzer, programs to be fuzzed need to implement an entry function with the following interface: `void run(char *input, size_t size)`. The fuzzer runs one test case by calling this function with the generated input.

3.1 Instrumentation for Coverage Maps

The main SBF instrumentation transforms the target program to maintain the distance map and the coverage map. It is implemented as an LLVM compiler pass, working on the intermediate representation (IR). This pass only modifies compare instructions. While the LLVM IR contains a `switch` instruction as well, we lower them to compares before carrying out the instrumentation.

After each compare instruction, we add a short code snippet that updates the distance and the coverage map. The distance map contains the distance values and the coverage map contains the hit counts for each edge. Each edge is assigned a unique ID. The distance and coverage maps are arrays, reserving one byte for each edge, indexed by the edge ID. The instrumentation after each comparison computes and stores the distance value of the non-taken edge, while it sets the distance to zero and increments the coverage counter for the taken edge.

3.2 Instrumentation to Identify Search Targets

We rely on dynamic data-flow tracking (DFT) for this instrumentation. LLVM already includes a DFT library, called DataFlowSanitizer, which can maintain and propagate multiple (taint) labels. We assign a separate label to each input byte index. After each comparison instruction, we check the labels associated with the arguments of the comparison. If there are associated labels, and the edge is not covered yet (as many times as we could from the current context), then we add the corresponding edge ID to the target list, along with the byte indices represented by the labels.

Note that DFT is needed only during the target identification phase. Since DFT has a significant runtime performance cost, we would like to avoid it while doing the (far more frequent) local searches. One approach for this is to create two binaries, one with DFT, and one without. Synchronizing and communicating between two processes however would add significant complexity and overhead. We therefore devised an alternative approach: we still create two versions of the targeted code, but link them into the same binary by renaming functions and other globals in one of them. Function versions that include DFT are prefixed with `dft_`. With this naming scheme, SBF calls `run()` during local search runs, and calls `dft_run()` for target identification.

3.3 Compiler Optimizations

We leverage compiler optimizations to create more edges, which enables finer granularity measurement of coverage. In particular, *Loop unrolling* and *function inlining* have the most potential for increasing the number of edges. Consider the following loop:

```
for (int i=0, i<3; ++i)
  if (!(isprint(input[i])))
    return 0;
return 1;
```

During compilation, this loop gets unrolled as follows:

```
if (!(isprint(input[0]))) return 0;
if (!(isprint(input[1]))) return 0;
if (!(isprint(input[2]))) return 0;
return 1;
```

In the original program there was only one input dependent comparison, but after unrolling, there are three. With this increase, SBF is able to differentiate more between different execution paths.

Function inlining helps in a similar manner. We can differentiate between the edges of the inlined function when they are called in different contexts, as there will be separate code for each context. Parser code often uses standard library functions for character or string comparisons, such as `isascii`, `isspace`, or `strcmp`, and `memcmp`. We also make sure that all these small C library functions are inlined. We collected the implementation of these typically single line functions from the `musl` C library and ensure that they get inlined when we compile a target with the SBF compiler.

3.4 The SBF Library

The SBF library contains the implementation of the SBF algorithms described in this paper, including the fuzzer cycle, the local search algorithms, the coverage check, etc., and also the `main` function. It is written from scratch, without using any external libraries and consists of about 6 KSLOC of C++.

Crashing signals are handled and state saving and restoring is also implemented. The state of the fuzzer consists of the last minimized test suite, the global coverage map and the statistics counters. These three are serialized when the process is stopped and restored when restarted. For more precise bug detection, optimally ASAN [38] or other LLVM sanitizer instrumentations can be used.

The hottest function in the code is the one comparing and updating the global coverage (C_{global} with C_{run}) after each run. We implemented an optimized version of this function using the Intel AVX2 instruction set to process the arrays 32 bytes at a time, which resulted in a ~2× speedup.

4 EVALUATION

We evaluate SBF by comparing it to three other test generation tools, namely to AFL [43], LibFuzzer [37], and KLEE [7]. AFL was the first fuzzer that demonstrated that coverage-guided fuzzing can be effective. LibFuzzer implements the same technique as AFL, but instead of forking a new process for running each test case, it does *in-process* fuzzing, by linking the fuzzer with the target library. As described above, we also follow this design with SBF. KLEE is the the most well known open source dynamic symbolic execution engine. We are first to provide an in-depth comparison of AFL, LibFuzzer and KLEE with each other on a range of programs.

We are most interested in the tools' effectiveness in finding bugs. Unfortunately, direct measurement of bug-finding ability is difficult. First, bugs are relatively rare in programs, which means that they provide a very low resolution metric. Second, it is inherently hard to measure the number of unique bugs found by a fuzzer. Fuzzing tools find crashes, but the number of crashes found does not equal the number of bugs found. To precisely establish the number of unique bugs given a set of crashes, one needs to do thorough root cause analysis, which requires a lot of manual effort and expertise.

Bugs, however, are just erroneous program states, and to be able to trigger them, a tool needs to be able to reach the corresponding program state. Therefore, the most important aspect of a bug finding tool is its ability to discover new program states, in other words, its ability to increase coverage. This is why the primary focus of our evaluation is the coverage reached by the tools. One could argue that even on identical code paths, tools may differ in terms of their ability to trigger bugs. However, in the case of SBF, we have not made any new contributions in triggering bugs: like other state-of-the-art fuzzers, SBF uses random blind mutations to trigger bugs, and/or sanitizers (e.g., ASAN [38]) to detect errors more precisely along paths that we can cover. This means that SBF's bug finding potential is closely correlated with its coverage. Nevertheless, we do provide direct results on SBF's bug finding ability, and compare it with the other tools. To deal with the sparsity problem and the unique bug identification problem, we rely on LAVA [12], a recently published tool that automatically inserts vulnerabilities into programs. LAVA can insert a high number of bugs in programs, each identified by a unique ID, which enables us to do a rigorous evaluation.

We also evaluated the contribution made by each of the three components of SBF to its coverage increasing ability. To this end we tested SBF in the four configurations summarized in Table 2. The first, CGF_{base}, is the most basic coverage-guided fuzzing algorithm, to which we add the SBF features one by one. For CGF_{base}, we run

	Inflation & Deflation	Targeting & mutation focus	Local Search
CGF_{base}			
$\text{SBF}_{\text{blind}}$	✓		
$\text{SBF}_{\text{targeted}}$	✓	✓	
SBF_{full}	✓	✓	✓

Table 2: Four SBF configurations used in evaluation.

Algorithm 1 with the search target identification and local search switched off, keeping only the random byte mutation phase. We repeat the main cycle without the test suite deflation and coverage reset in the outer loop. This means that we simply keep going through each test case in the test suite, and for each test case, we go through each byte and randomly mutate them.

In the next configuration, called $\text{SBF}_{\text{blind}}$, we enable the test suite inflation/deflation in the outer loop. With, $\text{SBF}_{\text{targeted}}$, we also add target identification. Here the local search function is also on, but it uses the RandomWalk algorithm, as defined in Algorithm 2. This means that the distance function feedback is not used at all. Essentially, we do the same random byte mutations, as in case of $\text{SBF}_{\text{blind}}$, but this time it is focused only to the bytes on which the targeted edge depends on. Finally, SBF_{full} is where all three features are switched on. In short, while SBF tells *what* bytes to mutate and *how*, $\text{SBF}_{\text{targeted}}$ only tells the *what*, while $\text{SBF}_{\text{blind}}$ tells neither.

4.1 Coverage Increasing Ability

We measured program coverage growth in time as SBF and the competing tools generated test cases. We picked four programs among the targets of the Google's OSS-Fuzz project [1]: two that takes textual inputs (`libxml` and `pcre`), and two that parse binary formats (`libpng` and `libjpeg`). We ran the four tools on the four different targets for 12 hours each. We repeated all experiments 4 times to also measure the variability of these randomized algorithms.

As SBF and LibFuzzer are in-process fuzzers, they require the definition of an entry function, i.e., the `run()` function described in Section 3, that calls the tested functionality. This typically means invoking the main parser function on the provided input buffer. Such functions were already available for our targets due to OSS-Fuzz. For testing with AFL and KLEE, we also linked a thin `main()` function to the targets. This takes a file as a command line argument, reads its contents and calls `run()` with it.

We used AFL and LibFuzzer with their default options and KLEE with the suggested set of flags provided on the tool's website, which were used for the `coreutils` experiments as well in their paper [7]. While the `coreutils` experiment used different symbolic input sizes for different targets, we set the symbolic input size to 64 bytes for all experiments. To match KLEE's input size, we set the maximum input size to 64 for the other three tools as well. Further, as KLEE cannot be provided with seed inputs, we started the test generation with an empty test suite with all tools.

We measured the coverage using `gcov`. We modified all competing tools to produce the same timing log as SBF, capturing the exact time of the generation of each new test case. We used this log to replay the created test cases on the `gcov` instrumented program, measuring the branch and line coverage as a function of time.

Figure 4 shows branch coverage for each tool normalized to the coverage reached by CGF_{base} at the given time. The x-axis shows

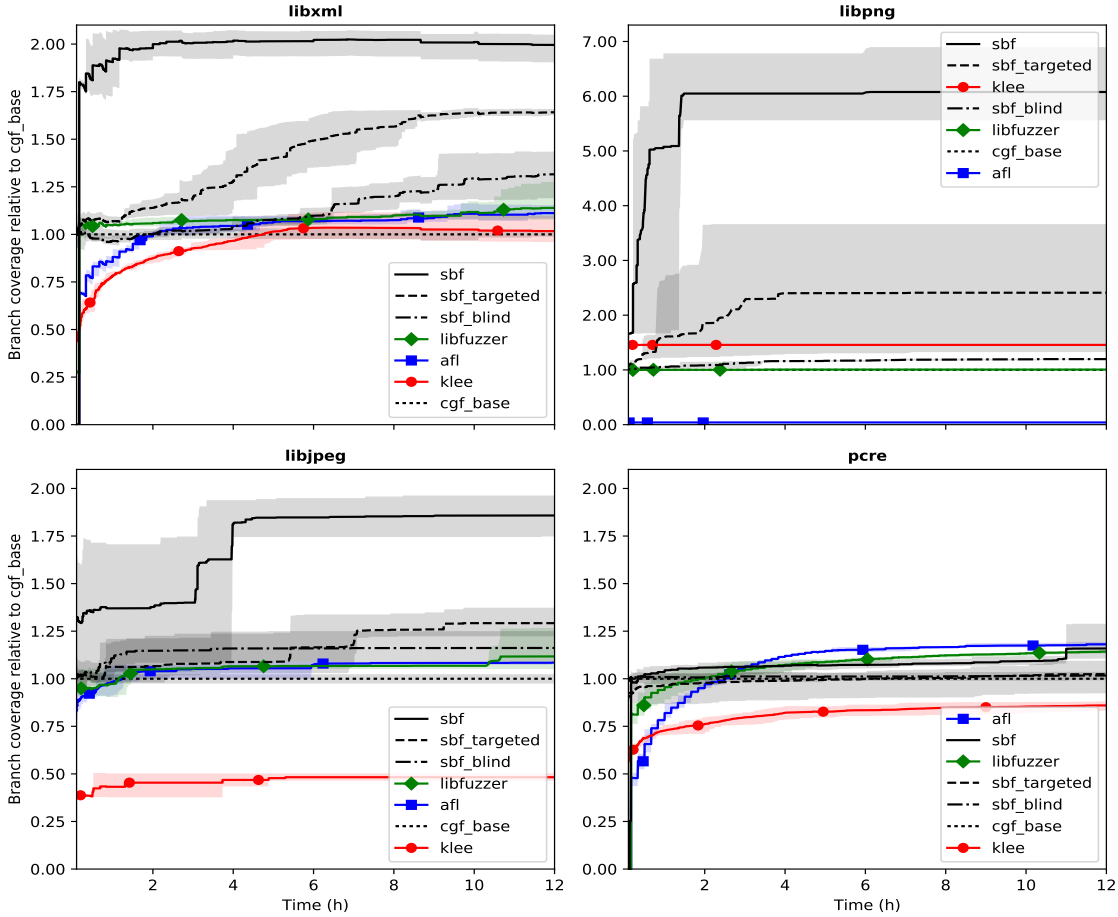

Figure 4: Coverage growth on four benchmarks, relative to base coverage guided fuzzing algorithm CGF_{base}.

the time the test generation tools ran, starting from 5 minutes to 12 hours. During the first five minutes, the relative behavior of different tools had not stabilized, so omit this period from the graph. On the y-axis 1 represents the coverage reached by CGF_{base}. The lines represent the mean of the four executions, while the shadows around the lines mark the area between the minimum and maximum values, to show variability.

In `libxml2` (a library for parsing XML documents) SBF obtains approximately twice as much coverage as the other tools. Note that it reaches significantly higher coverage in the first few minutes than what is achieved in 12 hours by the other tools. The three competing tools reach similar levels of coverage as CGF_{base}, with AFL and KLEE taking several hours to get there. SBF_{blind}, with only the inflation/deflation switched on, already outperforms all three competing tools. $SBF_{targeted}$, switching on focused mutation, further improves coverage by 50% after 12 hours. Finally with full-featured SBF, we can see the effect of the guided local search, with which we reach high coverage in a just few minutes. The XML format has a complex grammar, but SBF creates XML files with a wide variety of XML tokens and keywords in them.

On `libpng`, which is the official reference library for the Portable Network Graphics (PNG) format, CGF_{base}, AFL, LibFuzzer, and KLEE saturate after just a few minutes. They stop making progress at slightly different coverage levels. AFL gets stuck covering very little, LibFuzzer reaches the exact same coverage as CGF_{base}, while KLEE saturates somewhat higher. On the other hand, SBF reaches 6 times more coverage than the base algorithm and LibFuzzer, after less than two hours. The `libpng` library is a relatively hard target, as PNG is a complex file format. Still, SBF generates the proper header, with the necessary signatures and also internal "chunks" identified by different four letter codes, without any initial seed input. Considering the contribution of the SBF features, switching on the targeted, focused mutation already improves a lot on the effectiveness. As in `libxml`, the graph shows that adding the feedback of the stochastic local search has a major impact on effectiveness.

The `libjpeg` benchmark implements JPEG image handling functions, including image encoding, decoding, transforming, etc. The benchmark program runs the JPEG decode/decompression function on the input. The two coverage-guided fuzzers perform very similarly to CGF_{base}, while KLEE saturates with half of the coverage reached by the fuzzers. SBF reaches significantly higher coverage than the competing fuzzers in a few minutes, and around 1.8× as much coverage in a few hours.

All tools perform very similarly on the last benchmark, the Perl Compatible Regular Expressions (v2) library, or PCRE2 in short. SBF leads during the first hours, AFL and LibFuzzer catch up in the long run and they saturate at around the same level as SBF. KLEE progresses slower and covers less than the other tools. We believe the reason there is not much difference between the performance of the three fuzzers on this benchmark, is because the target is easily discoverable even with completely blind and random fuzzing. The tokens of a regular expression are typically single characters, which means that it is easy to generate a wide variety of expressions, just through random byte mutations.

4.2 Bug Finding Ability

Measuring the number of bugs found in programs with only a few bugs in them is not statistically robust. To directly evaluate the bug finding ability of tools, we need targets with a large numbers of bugs in them. LAVA [12] is a tool that enables such evaluation by automatically adding many realistic vulnerabilities to existing code. In order to evaluate SBF's and the other tool's bug finding power, we used one of the LAVA benchmarks, namely the LAVA modified `base64` utility that has 44 injected vulnerabilities in it. We ran all four tools on this target for 5 hours (same as the experiment in the LAVA paper [12]), and as before, we repeated the experiment four times, in order to get an average performance.

In order to make the `base64` utility testable with SBF and LibFuzzer, we replaced the original `main` of the program (that parses command line arguments) with a `run()` entry function that calls the Base64 decode function on the provided input buffer. With AFL and KLEE we tested the same code, linked with an additional `main()` function to read the input from a file and pass it to `run()`.

We ran two experiments, one where we ran the tools with an empty initial test suite, and one where a seed file was provided. The results are shown in Figure 5. We plot the percentage of the found bugs growing with time. Note that the time axis is in logarithmic scale. Without a seed input, only SBF and KLEE found bugs. SBF found all bugs in around 15 seconds in each run, while KLEE found 5 to 6 bugs each time, in 19 minutes on the average.

In the second experiment we provided the fuzzers with an initial Base64 encoded seed input (the one provided by the benchmark). KLEE is not shown on this plot as it does not accept seed inputs. This time both AFL and LibFuzzer were able to find bugs. AFL found 5 bugs in one of the runs, but no more than one in the other runs. On average, this is 2 bugs in 4 hours. LibFuzzer did better with an average of 22 bugs in 3 hours and 15 minutes. SBF performed similarly with or without a seed input, and found all injected bug in a matter of seconds.

The reason why SBF is so much more effective in finding bugs is that LAVA introduces a new edge for each vulnerability it injects. SBF targets each of these, and the directed local search algorithm can modify the relevant input bytes to trigger the targeted bug very quickly. In contrast, blind fuzzers do not target any particular edge and they carry out their mutations randomly, therefore it is only a matter of luck and time whether bugs are found.

5 RELATED WORK

Fuzzing. Black-box fuzzers [28, 34, 42, 45] run the target with randomly mutated or generated inputs without any feedback. If we already have a test suite with high coverage, these fuzzers can trigger bugs along the paths covered by this test suite, but they are not effective in increasing coverage. For this reason, modern fuzzers such as AFL [43] and LibFuzzer [37] are coverage-guided: whenever a mutant happens to cover a new path, it is selected for additional mutation. Using coverage to provide feedback in this manner, these fuzzers are able to increase coverage. They may even be able to operate without any initial test suite. Nevertheless, mutations themselves are carried out blindly. SBF, in contrast, incorporates a *directed* approach based on light-weight instrumentation to decide *what* to mutate, and *how* to mutate.

AFLFast [6] is a recent extension of AFL that finds new paths faster by applying a heuristic on which files to fuzz more. The intuition behind its strategy is that there is a better chance of reaching new edges from inputs that exercise rare paths, so they prioritize fuzzing those inputs more. This is a useful heuristic when the fuzzer has no information on what exactly are the reachable (touched) new edges from an input's path. With SBF, due to the target identification phase, we have this information, and we can avoid fuzzing inputs that do not touch any uncovered edge. Our targeting strategy inherently applies AFLFast's prioritization even more precisely, as we only fuzz inputs that have the potential to reach a new edge. The more reachable edges there are, the more we fuzz, as the number of identified edges determines the number of local searches.

Taint-guided fuzzers [5, 16, 23, 41] use data-flow tracking to decide which parts of a given input to mutate more than others. These tools use taint-tracking to identify what parts of the input flow into potentially dangerous operations, such as library function calls (e.g., `malloc` or `strcpy`), and fuzz those parts of the input more. This can help in triggering a bug earlier along a path covered by an existing test suite, but does not help in finding new paths. More importantly, they do not address the problem of *how* to mutate.

Dynamic Symbolic Execution. Dynamic symbolic execution [7–9, 17, 36], also called *white-box fuzzing* [18], is a systematic testing approach. It generates inputs in two steps. First, the program is run symbolically in order to extract a path condition formula. Then, as a second step, a constraint solver (SMT solver) is used to find input assignments satisfying formulas representing alternative paths. SBF achieves some of the directionality of DSE while avoiding the (costly and complex) symbolic execution step. Recently, a stochastic local search (SLS) based constraint solver [15] was shown to be competitive with commonly used (systematic) constraint solvers [4, 11]. The SLS based SMT solver of Fröhlich et al. [15] even outperformed the state-of-the-art CDCL based Z3 algorithm on some benchmark formulas that were extracted by the SAGE [18] DSE system. These results suggest that stochastic local search is up to the task of tackling typical program execution path constraints, and justifies its use in SBF to achieve directionality.

In the traditional DSE setting, a local-search based solver would try to find a satisfying solution for a formula by searching the input space with the guidance of a distance function, which is evaluated on the formula. SBF carries out a similar search too, but instead

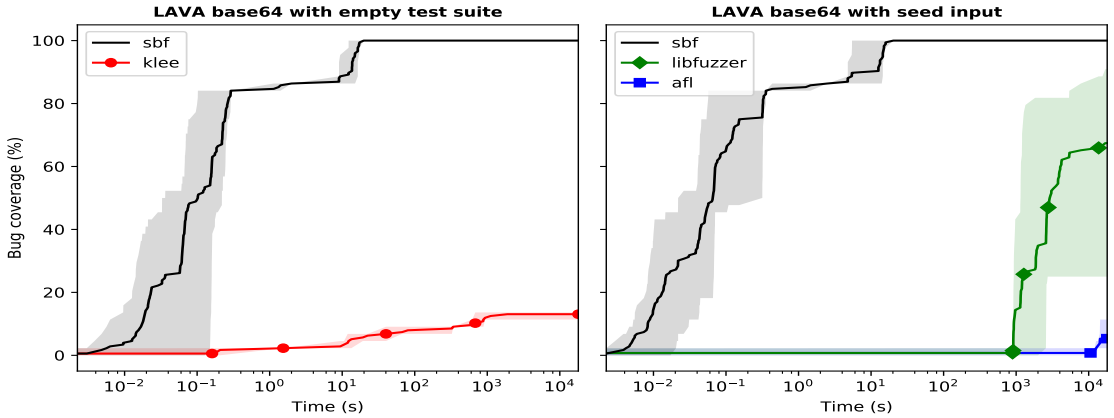

Figure 5: Bugs found in the LAVA base64 benchmark. Time is in logarithmic scale.

of evaluating the distance of an input assignment on an extracted formula, the distance is computed by executing the instrumented program. Moreover, while the constraint solver would try to satisfy the entire path constraint, we focus on just the last condition. This means that our approach is incremental in generating new inputs: it reuses the part of the existing input that is not influencing the targeted edge and mutates only the part that does. Although these mutations can sometimes cause the program to take an unintended path, our results suggest that the trade-off we make is very beneficial.

The main advantage of SBF, shared with fuzzers, is that it is not fixated on taking a particular path. In particular, if a new mutant that was intended to cover a certain edge ends up going down an unintended path, SBF does not discard it. Instead, if the mutant uncovers new behavior, then it is put back on the work list, and serves as a basis to further increase coverage. In contrast, an external constraint solver cannot make any use of solutions that may very well be useful for discovering new behavior, but do not satisfy the current constraint given to it.

Fuzzing+DSE hybrids. Hybrid approaches [24, 32, 40] can combine some of the benefits of fuzzing and DSE. In these systems, cheap and lightweight fuzzing is used to generate test cases until it gets stuck and does not find new paths any more. DSE is used at this point to "punch through" difficult conditions that the fuzzer was unable to get through. Instead of relying on such an ad-hoc approach for combining the benefits of fuzzing and DSE, SBF uses a more systematic approach by building directionality into fuzzing. Our results demonstrate that SBF's directionality helps it increase coverage at a much faster rate than existing systems.

Note, that SBF can also be used as component of a hybrid system in addition to blind fuzzers and DSE. Adding SBF to an existing Fuzzer+DSE hybrid will make the system as a whole more efficient, as more coverage targets can be reached with cheaper input generation. Namely, most "hard" paths for which symbolic execution was previously necessary, can be quickly covered by SBF.

Search-based software testing (SBST).. The idea of using local search algorithms for generating test data has been investigated for a long time [2, 26, 27, 29]. SBST has primarily been used for unit test generation. These tools typically establish the set of paths or branches that they want to reach statically and then target them individually. AUSTIN [21, 22] is one of the most advanced SBST based unit testing tool for C. It targets edges in functions using local search, but uses different scoring function and different search algorithm than SBF. For each edge in the function's control flow graph a new search is initiated. Searches are independent from each other, so the information we gained by generating an input that reaches one edge is lost or unused for generating input to reach another edge along the same path. In contrast, SBF targets edges incrementally, similarly to SAGE's generational search. Also, AUSTIN's search is intra-procedural, meaning that only the nodes inside the function are considered for evaluating the score, while SBF does not have such limitation.

Another example of a real-world tool using SBST is EvoSuite [14]. It is a unit test case generator for Java, using evolutionary algorithm. EvoSuite mutates and scores the entire test suite in one iteration, instead of trying to reach specific coverage targets with individual test cases. The test suite is scored based on the total number of covered edges. In each iteration step, the genetic algorithm treats the entire test suite as the population, and mutates it towards a better (population) score, instead of improving individual test cases.

Vuzzer [33] is a recent fuzzer that also uses evolutionary algorithm to prioritize its input mutation and uses coverage as a scoring function. It also assigns weights to basic blocks using a prior static analysis, and the score (fitness) is calculated using the weights of the covered blocks. This way it operates as a coverage-guided fuzzer but with more detailed coverage information feedback. SBF differs from these tools by targeting potentially reachable new edges using a local search.

6 CONCLUSION

In this paper we introduced *search-based fuzzing*, a new test generation technique that incorporates some of the directionality of symbolic execution with the simplicity and scalability of fuzzers. Our target identification algorithm lets us collect which nodes/edges are affected by which inputs so we can target those edges with stochastic local search effectively, with a reduced search space. We thoroughly explored the design space of local search algorithms, and

devised an algorithm that works well on broad set of benchmarks. Finally, our comparison based coverage metric and test suite inflation and deflation technique eliminates many of the limitations of the edge coverage metric used by existing coverage-guided fuzzers. SBF is an available tool that bridges the gap between directed and scalable test generators.

ACKNOWLEDGMENTS

REFERENCES

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. Google Testing Blog. (December 2016). https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html

[2] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. 2010. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *Software Engineering, IEEE Transactions on* 36, 6 (Nov 2010), 742–762. https://doi.org/10.1109/TSE.2009.52

[3] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, and Alex Rebert. 2016. Unleashing the Mayhem CRS. https://blog.forallsecure.com/2016/02/09/unleashing-mayhem/. (2016).

[4] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[5] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. 2012. A Taint Based Approach for Smart Fuzzing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. 818–825. https://doi.org/10.1109/ICST.2012.182

[6] Marcel Bohme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security, 2016.*

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

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

[9] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394. https://doi.org/10.1109/SP.2012.31

[10] Codenomicon. 2014. Heartbleed. http://http://heartbleed.com/. (2014).

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[12] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale Automated Vulnerability Addition. In *In IEEE Symposium on Security and Privacy (S&P)*.

[13] Joshua Drake. 2015. Stagefright: Scary Code in the Heart of Android. In *BlackHat USA*.

[14] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[15] Andreas Frohlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. 2015. Stochastic Local Search for Satisfiability Modulo Theories. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*. AAAI - Association for the Advancement of Artificial Intelligence. http://research.microsoft.com/apps/pubs/default.aspx?id=238374

[16] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

[18] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf

[19] Peter Goodman. 2016. A fuzzer and a symbolic executor walk into a cloud. https://blog.trailofbits.com/2016/08/02/engineering-solutions-to-hard-program-analysis-problems. (2016).

[20] Holger H Hoos and Thomas Stützle. 2004. *Stochastic local search: Foundations & applications*. Elsevier.

[21] Kiran Lakhotia. 2009. *Search-Based Testing*. Ph.D. Dissertation. King's College London.

[22] Kiran Lakhotia, Mark Harman, and Hamilton Gross. 2013. AUSTIN: An Open Source Tool for Search Based Software Testing of C Programs. *Inf. Softw. Technol.* 55, 1 (January 2013), 112–125. https://doi.org/10.1016/j.infsof.2012.03.009

[23] Guangcheng Liang, Lejian Liao, Xin Xu, Jianguang Du, Guoqiang Li, and Henglong Zhao. 2013. Effective Fuzzing Based on Dynamic Taint Analysis. In *Computational Intelligence and Security (CIS), 2013 9th International Conference on*. 615–619. https://doi.org/10.1109/CIS.2013.135

[24] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 416–426. https://doi.org/10.1109/ICSE.2007.41

[25] Felipe Andres Manzano. 2010. The Symbolic Maze! Feliam's Blog. (October 2010). https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

[26] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156. https://doi.org/10.1002/stvr.v14:2

[27] P. McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. 153–163. https://doi.org/10.1109/ICSTW.2011.100

[28] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (December 1990), 32–44. https://doi.org/10.1145/96267.96279

[29] W. Miller and D. L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Transactions on Software Engineering* SE-2, 3 (Sept 1976), 223–226. https://doi.org/10.1109/TSE.1976.233818

[30] Ducson Nguyen. 2013. Hybrid Concolic Execution. http://blogs.grammatech.com/hybrid-concolic-execution-part-1. (2013).

[31] P. Oehlert. 2005. Violating assumptions with fuzzing. *IEEE Security Privacy* 3, 2 (March 2005), 58–62. https://doi.org/10.1109/MSP.2005.55

[32] Brian S. Pak. 2012. *Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution*. Master's thesis. Carnegie Mellon University.

[33] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*. https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf

[34] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[35] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. *SIGPLAN Not.* 48, 4 (March 2013), 305–316. https://doi.org/10.1145/2499368.2451150

[36] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[37] Kostya Serebryany. 2015. Simple guided fuzzing for libraries using LLVM's new libFuzzer. http://blog.llvm.org/2015/04/fuzz-all-clangs.html. (2015).

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*. https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[39] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[40] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*.

[41] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*. 497–512. https://doi.org/10.1109/SP.2010.37

[42] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2508859.2516736

[43] Michal Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. (2014).
[44] Michal Zalewski. 2014. Bash bug: the other two RCEs, or how we chipped away at the original fix. https://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html. (2014).
[45] Mingyi Zhao and Peng Liu. 2016. Empirical Analysis and Modeling of Black-Box Mutational Fuzzing. In *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*. Springer International Publishing, Cham, 173–189. https://doi.org/10.1007/978-3-319-30806-7_11