IET Software

Research Article

IET Journals
The Institution of Engineering and Technology

Discover deeper bugs with dynamic symbolic execution and coverage-based fuzz testing

ISSN 1751-8806
Received on 14th August 2017
Revised 7th April 2018
Accepted on 7th May 2018
E-First on 30th May 2018
doi: 10.1049/iet-sen.2017.0200
www.ietdl.org

Bin Zhang[1,2], Chao Feng[1] ✉, Adrian Herrera[2], Vitaly Chipounov[3], George Candea[2], Chaojing Tang[1]
[1]School of Electronic Science and Engineering, National University of Defense Technology (NUDT), Changsha, Hunan, People's Republic of China
[2]School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
[3]Cyberhaven, Inc., 401 Park Drive, Suite 811 Boston, MA 02215, USA
✉ *E-mail: binzh4ng@hotmail.com*

Abstract: Coverage-based fuzz testing and dynamic symbolic execution are both popular program testing techniques. However, on their own, both techniques suffer from scalability problems when considering the complexity of modern software. Hybrid testing methods attempt to mitigate these problems by leveraging dynamic symbolic execution to assist fuzz testing. Unfortunately, the efficiency of such methods is still limited by specific program structures and the schedule of seed files. In this study, the authors introduce a novel lazy symbolic pointer concretisation method and a symbolic loop bucket optimisation to mitigate path explosion caused by dynamic symbolic execution in hybrid testing. They also propose a distance-based seed selection method to rearrange the seed queue of the fuzzer engine in order to achieve higher coverage. They implemented a prototype and evaluate its ability to find vulnerabilities in software and cover new execution paths. They show on different benchmarks that it can find more crashes than other off-the-shelf vulnerability detection tools. They also show that the proposed method can discover 43% more unique paths than vanilla fuzz testing.

1 Introduction

Fuzz testing is a popular technique for automatic software vulnerability detection [1–3], but suffers from low efficiency when applied to real-world software [4–9]. Software often parses complex input formats such as PDF, DOCX, or JPEG, which generates deep execution paths with complex conditions. Traditional random fuzzers generate shallow test cases because they are unable to guess the inputs that would help reach deeper parts of the code. More sophisticated fuzzers discard test inputs that do not add new coverage and keep the remaining inputs in a seed file queue. They then derive new test input from the seed queue using genetic algorithms [8–10]. Although coverage-based fuzz testing is able to discover more paths than traditional fuzz testing, it is nevertheless incapable of triggering bugs that are deeply nested in complex code areas, due to the random nature of the mutations.

Dynamic symbolic execution alleviates some of the challenges encountered by fuzzers. Whereas fuzzers try millions of different concrete inputs (e.g. ‘abc’, 1,...) in order to reach deeper parts of the code, symbolic execution uses symbolic inputs (e.g. λ) that concisely summarise all possible concrete values (e.g. all values of a 32-bit integer). When a symbolic value reaches a branch condition, the symbolic execution engine invokes a constraint solver in order to compute the exact value that would drive the program down the desired path. Unfortunately, pure symbolic execution often results in an exponential number of paths that bottlenecks the constraint solver.

Recent work aims to combine the advantages of symbolic execution and fuzz testing. In this hybrid approach [11–14], corner cases that are difficult for fuzzers to cover are generated from dynamic symbolic execution by solving the corresponding path constraints. Conversely, dynamic symbolic execution also benefits from the fuzzer-generated seeds in order to quickly reach more code areas without getting lost in a large execution tree. Driller, which is built on top of the Angr symbolic execution engine [15], and the AFL fuzzing engine [10], have attempted to leverage symbolic execution to solve the branches guarded by complex path conditions to avoid saturation of fuzzer [9]. Driller's performance in DARPA's cyber grand challenge [16] demonstrates the potential of these hybrid testing approaches.

In hybrid testing, such as Driller, the performance gain from dynamic symbolic execution is still limited by particular program structures, such as symbolic pointers and loops [4, 17–19]. These structures quickly generate many redundant paths that do not trigger new behaviours but result in *path explosion*. This is compounded by the possibly large number of seed files generated by the fuzzer.

This paper makes three main contributions. We propose three new techniques to improve the efficiency of hybrid testing. First, we improve the lazy concretisation of symbolic pointers (LCSPs) presented in [20]. Second, we enhance the AFL's loop bucketing technique [10] in order to avoid getting stuck in loops that have a symbolic iteration counter. Third, in order to address the large size of the seed queue, we propose a distance-based seed selection algorithm in order to improve coverage when testing time is limited. Each seed in the queue is weighted by runtime information, i.e. path and memory coverage, then the seed with greater weight is assigned with more mutation times. We also implemented our prototype BREACHER and benchmarked it on (i) a sample program that contains nine different types of representative bugs, (ii) the LAVA benchmark suite [21]. The experiments show that BREACHER triggers more bugs than other state-of-the-art vulnerability detection tools. The evaluation results on several real-world programs show that our approach can discover 43% more unique paths on average than traditional random fuzz testing. Also, BREACHER exposed several unreported crashes in real-world programs.

The rest of this paper is organised as follows. Section 2 introduces dynamic symbolic execution and hybrid testing. Section 3 presents the details of how we deal with *path explosion* caused by symbolic pointers and loops. Section 4 presents the distance-based seed selection algorithm. Section 5 describes the implementation of BREACHER and the evaluation results. Section 6 discusses the limitations of our work and possible further research topics. Finally, Section 7 reviews related work and Section 8 concludes the paper.

```
 1 int check(int magic, int checksum, int flag) {
 2    if (magic != 0xFFD8) {
 3       perror("Bad magic number!\n");
 4    }
 5    if (checksum != bswap_32(checksum)) {
 6       perror("Checksum incorrect!\n");
 7    }
 8    path_explosion(); // path explosion here
 9    if (flag == CRASH) {
10       abort(); // bug here
11    }
12    printf("Safe!\n");
13    return 0;
14 }
```

Fig. 1 *Motivating code for preliminaries. Note that all example code in this paper is given as source code for readability, but our approach operates directly on binary programs*

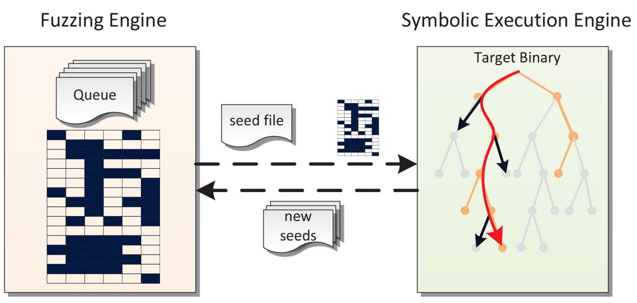


Fig. 2 *Dynamic symbolic execution assisted fuzz testing. The symbolic execution engine can help generate fresh seeds for the fuzzing engine based on the seed files and the already explored path information*

2 Preliminaries

In this section, we first review the theoretical foundation of dynamic symbolic execution. Then we provide a simple example in Fig. 1 to illustrate the concept of hybrid testing based on fuzz testing and dynamic symbolic execution.

2.1 Dynamic symbolic execution

Symbolic execution is an analysis method to determine what input can drive execution to specific code regions [22]. It interprets the program by assigning *symbolic data* rather than actual (*concrete*) data to the program inputs.

Consider a program P which consists of a set of program variables Var and a set of instructions $Inst$. An *execution path* is a serial of instructions, such as $I_0 \rightarrow I_1 \rightarrow \cdots \rightarrow I_n$, where $I_i \in Inst$. Unlike execution path, which focuses on instructions, a *program state* describes current execution state, which holds the value of each variable (a variable can be represented by either register or memory). Since the bits of register and memory are bounded, the number of program state can be enumerated. However, the total number of execution path may be infinite, which is also known as 'path explosion' problem.

Dynamic symbolic execution maintains a *symbolic state* $\mathcal{S} = \langle I, M, S, pc \rangle$, where $I \in Inst$ denotes the next instruction to be executed, M is the concrete memory store which maps Var to concrete data, S is the symbolic memory store which maps Var to symbolic expressions, and pc is the symbolic path constraint which is a first-order quantifier free formula over symbolic expressions. Since each symbolic state can be individually mapped to an execution path, we can use *symbolic state* to equivalently represent an execution path. In the following sections of this paper, all *states* denote the *symbolic state* unless explicitly stated. The 'path explosion' problem can also be expressed as 'state explosion'.

Under dynamic symbolic execution, program P operates on both concrete memory M and symbolic memory S by executing each instruction $I \in Inst$. The semantics for different types of instructions are listed in Table 1.

The objective of dynamic symbolic execution is to systematically explore all feasible paths of program P under the initial input. Consider an input vector ι which steers program P to execute a unique finite program trace $s_0 \xrightarrow{I_1} s_1 \cdots \xrightarrow{I_n} s_n$, where $I_1 \cdots I_n \in Inst$ and $s_1 \cdots s_n$ are program states. Then during the execution of dynamic symbolic execution, different types of instructions are handled differently according to their semantics. If I_i is an assignment instruction $v := e$, dynamic symbolic execution updates the symbolic memory of v directly with expression e, i.e. $S(v) := e$. If I_i is a conditional instruction as defined previously, any satisfying assignments to the Boolean expression $pc \wedge e$ will lead P to execute the *then* branch and any satisfying assignments to Boolean expression $pc \wedge \neg e$ will steer P to run the *else* branch. Dynamic symbolic execution tries to explore both *then* and *else* branches simultaneously by *state forking*. To fork a new state, dynamic symbolic execution uses a constraint solver T to generate solutions both for $pc \wedge e$ and $pc \wedge \neg e$ and updates the path constraints accordingly. By forking new program state for each executed conditional instruction, all possible path constraints can be enumerated and eventually all feasible paths in P can be exercised. Whenever dynamic symbolic execution executes a terminate instruction, it terminates the current state and employs the constraint solver to solve the current pc to generate a corresponding *test input*.

Table 1 Instructions in dynamic symbolic execution, as well as their expressions and semantics

Type	Expression	Semantic
assignment	$v := e$	update v with expression e
terminate	abort/exit	terminate current state
conditional	if e then I_{true} else I_{false}	fork current state

Dynamic symbolic execution is affected by the *path explosion* problem. Since dynamic symbolic execution may fork a new state for every conditional branch, the number of states may grow exponentially in the number of conditional instructions [23]. A large number of states will quickly exhaust computation resources and halt testing.

2.2 Hybrid testing

The sample program in Fig. 1 tries to verify the three input parameters. Hybrid testing starts from a coverage-based fuzzer which quickly generates an input that satisfies the *else* branch of the conditional instruction at line 2. However, verification of checksum at line 5 would typically prevent the fuzzer from going any further. When the fuzzer gets stuck, hybrid testing switches to dynamic symbolic execution engine to get more coverage.

Fig. 2 shows the general architecture of a hybrid testing approach based on fuzz testing and symbolic execution. The fuzzing engine performs coverage-based fuzz testing, and shares the already explored path information with the symbolic execution engine. This already explored path information can be recorded in any form. For example, AFL intercepts transitions between basic blocks as well as the branch-taken hit counts, and stores these information into a *Bitmap*. Then each test case in the seed queue will be sent to symbolic execution engine to disclose more new paths. Unlike traditional dynamic symbolic execution, the symbolic execution engine only forks for branches that have not been previously covered. For example, as shown in Fig. 2, suppose the red execution trace is the path that the fuzzer will take. In this case, the symbolic execution engine will only fork new states when it finds uncovered branches (the blue branches) according to the shared internal information from the fuzzer. Each forked state will generate a new test case which can help the fuzzer to reach new code areas. Based on this hybrid testing method, the verification of `header->checksum` at line 5 can be solved by symbolic execution engine and generate a new test case for further fuzzing.

However, function `path_explosion` makes it hard to solve the conditional instruction at line 9. For example, when using BFS exploration strategy, this function will quickly saturate the symbolic state's number budget that used to avoid memory overhead [9]; DFS exploration works better than BFS in this example, but it may degrade to exhaustive testing in some cases

```
 1 // An example to demonstrate the symbolic ↩
       pointer problem.
 2 #define F 0 /* never appears in text */
 3 #define T 1 /* in plain ASCII text */
 4 #define I 2 /* in ISO-8859 text */
 5 #define X 3 /* in non-ISO extended ASCII*/
 6
 7 const char text_chars[256] = {
 8  F, F, F, F, F, F, F, T, T, T, T, F, T, T, F, F
 9  /* 0x1X ... 0xeX list here */
10  I, I, I, I, I, I, I, I, I, I, I, I, I, I, I, I
11 }
12
13 int looks_ascii(const uchar *buf, size_t nbytes↩
       , uchar *ubuf, size_t *ulen)
14 {
15    size_t i;
16    *ulen = 0;
17    for (i = 0; i < nbytes; i++) {
18       int t = text_chars[buf[i]];
19       if (t != T)
20          return 0;
21       ubuf[(*ulen)++] = buf[i];
22    }
23    printf("ascii %" SIZE_T_FORMAT "u\n", ulen);
24    if (ubuf[0] == 'D' && ubuf[1] == 'E' &&
25       ubuf[2] == 'A' && ubuf[3] == 'D') {
26       assert(0 && "Bug Triggered!");
27    }
28    return 1;
29 }
```

Fig. 3 *Motivating code derived from file in GNU Coreutils that contains symbolic pointer dereference at line 18*

(e.g. when `path_explosion` deals `flag` as a symbolic pointer). We will discuss our mitigation for this problem in Section 3.

3 Mitigating path explosion in Symbolic Path Finder (SPF)

Hybrid testing can help to reduce *memory overhead* by limiting the number of states to an acceptable level. However, as mentioned in Section 1, symbolic pointers and loops will quickly generate lots of useless states that may not cover new code areas but bring serious performance overhead.

To address the large number of states forked from symbolic pointers, we propose a novel LCSP which can not only reduce the number of states but also improve coverage. For symbolic loops, we introduce an optimisation based on AFL's *loop bucket* to control forking in symbolic loops. By doing this, execution can reach deeper code areas without generating lots of states. Both improvements will be discussed in the following sections.

3.1 Lazy concretisation of symbolic pointer

The code snippet in Fig. 3 shows the basic symbolic pointer problem in dynamic symbolic execution. The first parameter (i.e. `buf`) of function `looks_ascii` points to memory that contains symbolic input data. The `nbytes` parameter is a concrete value that denotes the size of the memory buffer pointed by `buf`. `ubuf` is a shadow buffer which is used for further processing. Function `looks_ascii` tries to determine whether each character of the symbolic input data appears in plain ASCII text, and returns immediately once a non-plain ASCII text character appears. The symbolic execution engine faces the symbolic pointer problem when executing the code at line 18 because `buf[i]` is from the symbolic input data. The memory range of `text_chars[buf[i]]` spans from `&text_chars` to `&text_chars+255`. Since the binary executable loses the type information, the symbolic execution engine may need to explore all 256 possible values for each `buf[i]` at the worst case. Meanwhile, the loop from line 17 to line 22 makes the trigger of the bug at line 26 even harder since more states will be forked when `nbytes` is a larger one.

There are different approaches for handling path explosion caused by symbolic pointers. For example, treating memory address as *fully symbolic* enables the executor to reason about all possible values for symbolic pointer [24–27]. This can be achieved by either forking states or employing nested *if-then-else* formulas which encode all possible values. However, since a symbolic address may point to any memory cell, fully symbolic memory model fails to scale for real-world binary software. Some researches leverage the *theories of arrays* to make fully symbolic memory model scalable [28, 29]. For example, KLEE [28] forks states for values that reference different objects, and the *theories of arrays* is leveraged within the same object. However, since our target is to analyse binary executable whose object size of data structure is unavailable, the number of objects increases because each possible value may reference to a different object. In contrast to reason about all possible values, a *partial symbolic* memory model has been proposed [15, 30, 31]. The partial symbolic memory model tries to concretise all symbolic pointer write operation and treats symbolic pointer read operation using a fully symbolic memory model when contiguous interval of possible values is small enough. However, if the possible values span a large area, partial symbolic memory model still needs to concretise the symbolic pointer which may lose some soundness paths.

The *lazy forking* strategy leveraged in S2E was proposed to avoid maintaining expensive symbolic pointers and ease the large number of states by forking *pending states* in concolic execution [20]. Consider a memory dereference instruction $I \in Inst$ in program P, suppose I tries to access memory indexed by a symbolic expression e_{addr}. Lazy forking treats such instruction I as a conditional instruction and forks new states for I. It first evaluates the concrete value as c_{addr}, then it constructs an equal expression $condition := EQ(e_{addr}, c_{addr})$ which points e_{addr} to this concrete value. Then it forks a new state $s_p = fork(s, \neg condition)$ which is labelled as a 'pending state'. After that, each possible value of e_{addr} will be exercised by systematically repeating this process. Even though lazy forking still needs to enumerate all possible values, it can avoid overhead by significantly reducing the total number of states that simultaneously exist in system. Here, the *condition* is called a *hard constraint* and $\neg$*condition* is called a *soft constraint*.

For example, for the memory dereference instruction at line 18 in Fig. 3, suppose the concrete value of `buf[i]` for $i \in [0, 1, 2, 3]$ is '`A`'. In this case lazy forking will fork a new pending state and add the soft constraint $\text{buf[i]} \neq$ 'A' to it. Meanwhile, the path constraint of the original state will be appended with the hard constraint $\text{buf[i]} =$ 'A'. By doing this, the 'path explosion' problem is postponed to a later moment.

However, the hard constraint may reduce the suffix feasible paths to a very small group. For example, suppose the address of a symbolic pointer can be expressed as $e_{addr} = f(v_1, v_2, \ldots, v_n)$, where $v_1, v_2, \ldots, v_n \in Var$ are variables of program P. Then expression $condition := EQ(e_{addr}, c_{addr})$ will limit the current execution path only feasible when $(v_1, v_2, \ldots, v_n)$ equals to $(c_1, c_2, \ldots, c_n)$, where c_i are the corresponding concrete values. Take the sample code in Fig. 3. The execution path to line 26 will be infeasible because the hard constraint limits the value of `buf[i]` ($i \in [0, 1, 2, 3]$) to '`A`'. The crash can only be triggered after enumerating all possible values for `buf[i]` ($i \in [0, 1, 2, 3]$) in the worst case since lazy forking still belongs to DFS state exploration strategy. So even though lazy forking can ease the path explosion problem, it may still need to take longer time to trigger interesting paths. This will result in performance loss, because the symbolic execution engine may hold up the fuzzer.

To mitigate this problem, we introduce a novel method LCSP which is built on top of lazy forking. The detailed algorithm of LCSP is shown in Algorithm 1 (see Fig. 4).

Whenever the execution engine touches a symbolic pointer e_{addr}, it obtains the range of all possible values $\mathscr{R}$ by invoking function `getRange` in the constraint solver. Then all memory values within the range are dissolved into buckets $\mathscr{B} = \langle v, addr \rangle$, where v is the memory value; $addr$ is the set of memory cell's address whose memory value is v. After this, we reuse S2E's lazy forking method to pick up the bucket that contains the concrete value of this symbolic pointer. For example, when executing the code at line 18 in Fig. 3, the range of the symbolic pointer is $\mathscr{R} = \&\text{text\_chars} + \{0, 1, \ldots, 255\}$. By scanning the memory cells within $\mathscr{R}$, we can build four buckets, i.e.

$$\mathscr{B}_0\colon \{v = F \mid addr = [0,1,2,3,4,5,6,\ldots] + \&\text{text\_chars}\}$$

$$\mathscr{B}_1\colon \{v = T \mid addr = [7,8,9,10,12,13,\ldots] + \&\text{text\_chars}\}$$

$$\mathscr{B}_2\colon \{v = I \mid addr = [160,161,\ldots,254,255] + \&\text{text\_chars}\}$$

$$\mathscr{B}_3\colon \{v = X \mid addr = [128,129,130,\ldots,159] + \&\text{text\_chars}\}$$

Since e_{addr}'s concrete value belongs to $\mathscr{B}_1$, we then introduce a new symbolic variable v_p into the engine and update current path constraint by adding expanded hard constraint *condition* to it. *Condition* is described as $P_v \cap P_p$, where

$$P_v = \{v_p == \mathrm{T}\};$$

$$P_p = \{e_{addr} == 0x41 + \&\text{text\_chars}\}\,.$$

During the following execution, all path conditions that related to the newly introduced symbolic variable v_p will be collected as pointer dereference constraint C_{pd}. This constraint is used to keep execution consistency.

When performing lazy forking, all the states whose path constraints contain the soft constraints will be collected into *Pending States* as well as the buckets. These pending states are grouped by the program variables (e.g. each byte in an input file) that affect the corresponding soft constraint. They are also ordered along with the execution trace. Then when the dynamic symbolic execution engine detects an infeasible branch due to hard constraints from symbolic pointer (lines 1–3), the branch condition C_F will be investigated to extract the program variables *offs* (line 4).

Since we have introduced new symbolic variable when dereferencing a symbolic pointer, we need to make sure *all conditions in C_{pd} are satisfied so that the generated test case can keep execution consistency*. Line 14 collects all satisfied buckets into $\mathscr{B}_{sat}$. The collect procedure is light-weight since `getSATBuckets` only needs to evaluate C_{pd} under each bucket's key (i.e. $\mathscr{B}.v$). For each satisfied bucket, real solutions for current e_{addr} are sieved out (lines 16–27). This is achieved by evaluating each *addr* in $\mathscr{B}_{sat}$ (lines 21 and 22). All C_{new} evaluated as True will be collected together (lines 23–25). The $\cup$ in line 24 compacts consecutive *addr*s into a range expression. For example, $\{e_{addr} == 4\}$, $\{e_{addr} == 5\}$, and $\{e_{addr} == 6\}$ will be transformed to $\{4 <= e_{addr} <= 6\}$.

By analysing all pending states that related to off_{fork}, the final extra condition *extraCond* is constructed (line 28). After this, we remove the hard constraints C_H (related to off_{fork}) and C_{pd} from a newly cloned state s_{tmp} from current state and append the final extra condition to it (lines 31–33). The reason why C_{pd} is stripped is because it is already satisfied at line 14. After appending all related conditions, the dynamic symbolic execution engine will try to generate a new test case (line 34), and once the generation successes, the test case t_{lsp} will be sent to the fuzzer to find more paths.

For the code snippet in Fig. 3, suppose `nbytes` is 5. Then when dynamic symbolic execution reaches line 24, there will be six states in the system: one execution state S_0 and five pending states (P_0, P_1, P_2, P_3, and P_4). P_i is forked when dereferencing `buf[i]` at line 18. The pointer dereference constraint C_{pd} is $\{(v_0 == T) \cap (v_1 == T) \cap (v_2 == T) \cap (v_3 == T) \cap (v_4 == T)\}$, where v_i is introduced for each symbolic pointer dereference at line 18; S_{C_H} is $\{C_{H0} \cap C_{H1} \cap C_{H2} \cap C_{H3} \cap C_{H4}\}$, where C_{Hi} is $\{v_i == \mathrm{T} \cap \mathrm{buf[i]} == '\mathrm{A}'\}$. Here, `buf[i]==‘A’` is derived from $\{\text{text_chars} + \mathrm{buf[i]} == \text{text_chars} + '\mathrm{A}'\}$Due to the hard constraint, the four condition instructions at line 24 are infeasible. We pick up the first failed condition $C_F = \{\mathrm{ubuf[0]} == '\mathrm{D}'\}$ to explain how LCSP works in detail.

The fork failure raises because line 18 introduces $\mathrm{buf[0]} = =' \mathrm{A}'$ in C_{H1}, which conflicts with condition

```
Input: Current state $S$, pending states $S_P$, failed condition
       $C_F$, the set of hard constraints $S_{C_H}$, pointer
       dereference constraint $C_{pd}$.
Output: Testcase $t_{lsp}$ if success.
1  if $C_F$ is not result from symbolic pointer then
2      return $null$;
3  end
4  $offs = S$.getInputOffset($C_F$);
5  $C_H = S_{C_H}$.getRelated($offs$);
6  $extraCond \leftarrow True$;
7  foreach $s_P$ in $S_P$ do
8      $off_{fork} = s_p$.getInputOffset($s_p$.forkCondition);
9      if $off_{fork}$ not in $offs$ then
10         continue;
11     end
12     $\mathcal{B} = s_p$.buckets;
13     $e_{addr} = s_p$.getForkAddr();
14     $\mathcal{B}_{sat} = s_p$.getSATBuckets($C_{pd}$, $\mathcal{B}$);
15     $curExCond \leftarrow False$;
16     foreach $\mathcal{B}$ in $\mathcal{B}_{sat}$ do
17         foreach $addr$ in $\mathcal{B}.addr$ do
18             $s_{tmp} = S$.clone();
19             $s_{tmp}$.getConditions().strip($C_H$);
20             $s_{tmp}$.addConstraint($C_F$);
21             $C_{new} \leftarrow \{e_{addr} = addr\}$;
22             $success = s_{tmp}$.evaluate($C_{new}$);
23             if $success$ then
24                 $curExCond \leftarrow curExCond \cup C_{new}$;
25             end
26         end
27     end
28     $extraCond \leftarrow extraCond \cap curExCond$;
29 end
30 $s_{tmp} = S$.clone();
31 $s_{tmp}$.getConditions().strip($C_H$);
32 $s_{tmp}$.getConditions().strip($C_{pd}$);
33 $s_{tmp}$.addConstraint($extraCond$);
34 $(success, t_{lsp}) = s_{tmp}$.generateTestCase();
35 if $success$ then
36     return $t_{lsp}$;
37 end
38 return $null$;
```

Fig. 4 *Algorithm 1: Lazy concretisation of symbolic pointer*

`ubuf[0]==‘D’`. According to Algorithm 1 (Fig. 4), LCSP first checks the input offset that results in this infeasible condition and deals all pending states related to this offset. Based on this, only P_0 is chosen to break `ubuf[0]==‘D’`, and the corresponding C_H and e_{addr} in Algorithm 1 (Fig. 4) are C_{H1} and `text_chars+buf[0]`, respectively.

Then all satisfied buckets of P_0 are sieved out by evaluating the pointer dereference constraint C_{pd}. There are four buckets for P_0 as mentioned before, i.e. $\mathscr{B}_0$, $\mathscr{B}_1$, $\mathscr{B}_2$, and $\mathscr{B}_3$, whose memory value v are $\mathscr{B}_0 \cdot v = F$, $\mathscr{B}_1 \cdot v = T$, $\mathscr{B}_2 \cdot v = I$, and $\mathscr{B}_3 \cdot v = X$, respectively. After evaluating all these four buckets, $\mathscr{B}_{sat}$ is $\{\mathscr{B}_1\}$ since C_{pd} is only satisfied when $v_0 = \mathscr{B}_1 \cdot v = T$.

After this, Algorithm 1 (Fig. 4) evaluates each *addr* in $\mathscr{B}_1$ to build the final extra condition (lines 16–27). The final constructed extra condition *extraCond* in this example is $\{e_{addr} == \&\text{text\_chars} +' \mathrm{D}'\}$. This *extraCond* keeps not only C_F but also C_{pd} be satisfied. Then LCSP fixes current path condition by stripping unrelated conditions and adding $\{e_{addr} == \&\text{text\_chars} +' \mathrm{D}'\}$ to it, where e_{addr} is `&text_chars +buf[0]`.

At last, LCSP invokes the constraint solver to generate a fresh test case. Here, after breaking condition `ubuf[0]=='D'`, the generated test case t_{lsp} is `DAAAA`. This test case will be sent to the fuzzer and the remained three conditions at line 24 will be solved in the same way. Thus, based on this algorithm, we can generate at least one fresh test case that satisfies the branch condition whenever a branch is infeasible because of lazy forking.

```c
 1 /* An example for symbolic loop */
 2 int verify_packet(PACKET* packet)
 3 {
 4    int length = getLength(packet);
 5    // Checking the end descriptor.
 6    while (--length) {
 7       if (read_byte(packet) == 0xFF) {
 8          return 0;
 9       }
10    }
11    char end = read_byte(packet);
12    return (int)(end == 0xFF);
13 }
```

Fig. 5 *Motivating example to demonstrate path explosion raised by symbolic loops*

```
   Input: Configured Loops $L$, Current Edge $CE$ and Bitmap $B_p$
   Output: Generated test cases $t_{slb}$
 1 if not $IsaLoopCycleEdge(L, CE)$ or not $IsaSymLoop(CE)$ then
 2     return;
 3 end
 4 $loopTimes = 1$;
 5 $UBs = ParseUncoveredBuckets(B_p)$;
 6 while TRUE do
 7     foreach $ub$ in $UBs$ do
 8         if $loopTimes$ within $ub$ then
 9             $t_{slb}.add(GenerateTestcase())$;
10             $UBs.remove(ub)$;
11         end
12     end
13     if $UBs$ is null then
14         return $t_{lsb}$;
15     else
16         $ExecuteOneCycle()$;
17         $loopTimes += 1$;
18     end
19 end
```

Fig. 6 *Algorithm 2: Symbolic loop bucket*

3.2 Optimisation for symbolic loop

Symbolic loop, whose loop control variable depends on symbolic data, is another common cause of path explosion since its loop times may range from 0 to infinite theoretically. Even though the hybrid testing method can ease path explosion, the states forked from a symbolic loop will quickly force the number of states to increase to the budget's upper bound.

The code snippet in Fig. 5 demonstrates this problem. Function `verify_packet` reads the `length` of the raw data from the packet at line 4. Then from lines 6 to 10, it investigates each bytes in the raw data to determine whether there exists the ending descriptor (i.e. `0xFF`) through a loop structure. Suppose we have a test case from the seed queue of the fuzzer and its `length` is `0xAA`, then the symbolic loop from lines 6 to 10 (`length` is symbolic) will result in path explosion. As the possible value of `length` is in the range of $[0, 2^{32} - 1]$, 2^{32} states will be forked from line 6 in the worst case. Most of the forked states from line 6 may not contribute to any new code coverage but only bring performance overhead. It is therefore important to also handle symbolic loops.

AEG [32] proposes *loop exhaustion* search strategy to handle symbolic loops. It tries to execute the loop as many times as possible. Since it focuses only on maximising the loop iterations which may fork lots of states, it thus may stuck the fuzzer from exploring more paths in our hybrid testing framework.

A *boundary state prioritisation* method has been proposed recently to ease the path explosion problem due to symbolic loops. The key idea of this prioritisation is to defer the analysis of uninteresting states based on the likelihood of a security vulnerability [33]. Specifically, it focuses only on three types of states for a symbolic loop: *no loop execution*, *single loop execution*, and *the largest number of loop executions*. The author implemented such a strategy within S2E [20] and produced a vulnerability detection tool *CAB-Fuzz*. It successfully found 21 undisclosed unique crashes in Windows 7 and Windows Server 2008 [33].

We extend this boundary state prioritisation method by integrating it with the *Loop Bucket* mechanism employed in AFL [10] to achieve better performance on finding vulnerabilities. AFL utilises *Loop Bucket* to avoid collecting too many test cases which only affect the loop times into the seed queue [10]. It groups the loop times into eight different buckets, i.e. [1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+]. Only changes that occur between different buckets will be regarded as new behaviours. Based on this idea, we proposed a *symbolic loop bucket* (SLB) to handle the symbolic loop when performing hybrid testing. The algorithm of SLB is described in Algorithm 2 (see Fig. 6).

Loops are extracted from the target program by static analysis. These loops will be configured in the dynamic symbolic execution engine to help it recognise loops in runtime. All the symbolic loops can be distinguished from the others by checking whether the loop exit condition is affected by symbolic data or not (lines 1–3). For the edge belongs to symbolic loop, the uncovered loop buckets for this loop will be obtained by analysing the *Bitmap* mentioned before (line 5). In already covered loop buckets, the program will loop for one more time without forking new state (lines 16 and 17). Once an uncovered bucket is reached, the corresponding test case will be generated and then this uncovered bucket will be removed from the uncovered loop buckets to avoid generating multiple test cases (lines 7–12). After generating test cases for all the uncovered buckets, the loop will be prohibited from being executed for more times. This can make sure that all the loop buckets will be covered without causing path explosion.

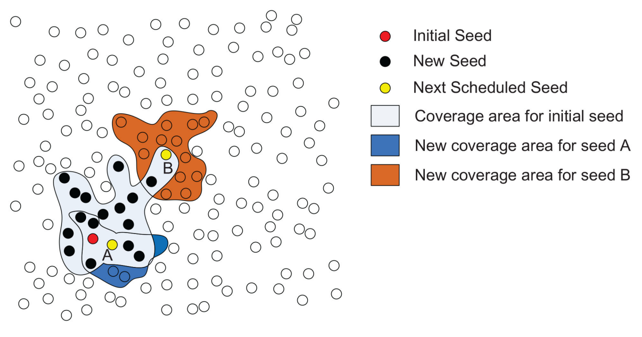
The symbolic loop is presented in Fig. 5. Suppose previous test cases have covered the buckets of [1], [2], [3], and [4–7]. Then *UBs* at line 5 in Algorithm 2 (Fig. 6) will consist of [8–15], [16–31], [32–127], and [128 +]. $loopTimes = [1, 2, \ldots, 7]$ will not fork any new states according to line 15 to 18. Then once the *loopTimes* reaches 8 which belongs to an uncovered bucket [8–15], the engine forks a new state, generates the corresponding the test case, and removes bucket [8–15] from *UBs*. The execution engines will not fork new states until *loopTimes* reaches 16, 32, and 128. Once all the loop buckets are covered, the forking in this symbolic loop will be disabled. It will continue cycling until *loopTimes* reaches the concrete value of length (i.e. 0xAA) and then exercise deeper code areas.

4 Distance-based seed selection

As previously discussed, the size of the fuzzer's seed queue will quickly reach a large number (especially with the assistance of dynamic symbolic execution) when testing large-scale modern software. The seed queue should therefore be rearranged to find more paths in a given time budget.

We realised that different seed files in the queue have different effects on path discovery. For example in Fig. 7, suppose each input that triggers new behaviour will be mapped to a specific dot. The fuzzer starts from an initial seed which is marked as red in this figure. The mutation results of this initial seed cover the black points as well as seed A and seed B. Then, in evolutionary fuzz testing, a test case will be picked up as the seed file for next round of mutation. Consider the choice between seed A and B in this figure. The *distance* (e.g. Euclidean distance) from the initial seed file to A is much shorter than B. This means that A is more *similar* to the initial seed file than B. So choosing A as the next seed file to mutate will only trigger two new paths (the blue area) as most of the mutated test cases are overlapped by what the initial seed has mutated (the grey area). However, because seed B has a longer distance from the initial seed than seed A, the coverage area of B (the brown area) will have less overlap by that of initial seed file than A. So choosing seed B will trigger more new behaviours than selecting seed A.

Based on this insight, we propose a seed selection method for the fuzzer based on the *distance* between test cases. Our method first maps each seed in the queue as a numerical vector. Then each seed is assigned with a *weight* value according to the distance from other test cases and its runtime information, which is then utilised



B
A
Initial Seed
New Seed
Next Scheduled Seed
Coverage area for initial seed
New coverage area for seed A
New coverage area for seed B

Fig. 7 *Motivating example to demonstrate how seed selection affects the testing efficiency*

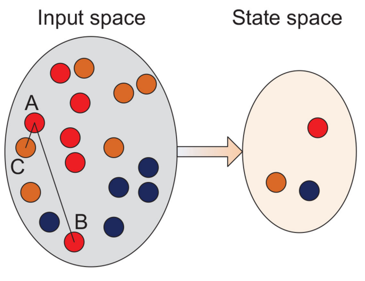


Fig. 8 *Illusion for distance. The distance from A to B is longer than from A to C in input space, which is opposite in state space because A and B will execute the same path (the red one)*

as the criteria to determine how many times a seed should be mutated. We investigate three popular distance measures, i.e. *Euclidean distance*, *cosine similarity*, and *Jaccard index*, as the distance metrics for calculating the *weight* value for each seed.

In order to measure the distances, all test cases should be mapped as numerical vectors. The mapping can be performed into either the input space or state space. Mapping into the input space cannot reflect the real relationship between test cases when considering the behaviour of the target program. As shown in Fig. 8, multiple different inputs can steer the program to execute the same path. For example, considering the three test cases namely `A.jpeg`, `B.jpeg`, and `C.jpeg` in this figure, suppose the corresponding contents are shown as follows:

A.jpeg: \xFF\xD8\xAA\xBB\xCC\xDD
B.jpeg: \xFF\xD8\xDD\xCC\xBB\xAA
C.jpeg: \xFE\xD8\xAA\xBB\xCC\xDD

If we calculate the *Euclidean distance* between these three files directly in the input space, ED_{AB} will be greater than ED_{AC} because there are more different bytes between A.jpeg and B.jpeg than that of `A.jpeg` and `C.jpeg`. This means `A.jpeg` and `C.jpeg` are more similar (as shown in Fig. 8) from the viewpoint of input space. However, for most of JPEG process programs, `A.jpeg` and `B.jpeg` will execute the same path, and `C.jpeg` will execute another path because `C.jpeg` is an illegal JPEG file (bad magic number). So from the viewpoint of the program, `A.jpeg` and `B.jpeg` are more similar than `A.jpeg` and `C.jpeg` even though `A.jpeg` and `C.jpeg` have only one bit difference. Since our objective is to maximise the coverage in the state space, we choose to map all the test cases as numeric vectors in the state space to calculate the distance.

In [34], all test cases are represented as a branch coverage vector $\boldsymbol{V} = (v_1, v_2, \ldots, v_N)$, where v_i is 0 means the branch is covered, otherwise 1. However, different test cases can affect different number of branches, so the mapped vectors may have different lengths which cannot be used directly for distance calculation. Meanwhile, it will be difficult to construct such vectors because obtaining all the branches and listing them *orderly* in each vector to avoid obfuscation between vectors is nearly impossible for binary programs. In AFL [10], the execution path information of each test case is stored as a *Bitmap*. By checking the hit count of this bitmap, AFL determines whether some new behaviours are triggered. So as this bitmap contains enough information to reflect the characteristics of a test case from the viewpoint of the state space, we choose the bitmap as our mapped vector to mitigate the costly mapping in [34].

Based on the mapped bitmap vectors, the test case queue in the fuzzer is enhanced by assigning each test case with weight W, where W is obtained by calculating the distance between every two test cases. Whenever a new seed file is found, the distance between this seed file and all the other files in the queue will be measured to calculate W. Meanwhile, the weight of all the other files in the queue will be updated according to the distance to the new seed file.

Rather than selecting the seed file that has the longest distance to the current seed file (which is only a *local optimum solution*), our search method selects the file that takes the longest average distance from all the other test cases as the next seed file. By doing this, we can achieve a *global optimum solution* for this searching problem. The weight W for test case t_k is defined as follows:

$$W_k = \frac{1}{N}\sum_{i=0}^{N} D(t_i, t_k)$$

where $D(t_i, t_k)$ denotes the distance between t_i and t_k based on the three distance measures mentioned before, and N is the size of the test case queue.

We also noticed that even though two test cases have the same W value, they may have different power on finding new paths. This is because path coverage is not the only criteria for testing, and other runtime information, such as memory operations, can also be leveraged to prioritise test cases. So we enhanced the weigh W with memory coverage to achieve better prioritisation result. The enhanced weight $\hat{W}_k$ for test case t_k is defined as the following formula, where $M(t_k)$ denotes the number of memory cells that t_k covers in byte

$$\hat{W}_k = M(t_k) + \frac{1}{N}\sum_{i=0}^{N} D(t_i, t_k)$$

AFL introduces *Energy* (i.e. mutation times) for each seed based on some properties like bitmap coverage, file size, execution time, and so on. For example, a seed with more bitmap coverage will be fuzzed with more times. We leveraged the similar idea in our distance-based seed selection. Based on the enhanced weight, our fuzzer engine assigns each seed with an extra energy to mutation. That is, a seed in the queue with greater weight will be fuzzed with more times. By assigning more mutation times for the seed that may cover more untouched code areas, we increase the probability to trigger more paths.

5 Implementation and evaluation

Breacher is built on top of AFL [10], a popular state-of-the-art genetic fuzzing framework, and the S2E symbolic execution platform [20]. S2E is a dynamic binary analysis platform which utilises selective symbolic execution to analyse whole software stacks at runtime. S2E reuses parts of the QEMU virtual machine [35], the KLEE symbolic execution engine [28], and the LLVM toolchains [36].

Our implementation consists of two main components, namely *SPF* and *Seacher*. The *SPF* component is leveraged to help the fuzzer dive into deeper code areas that are guarded by complex path constraints. Techniques to handle the *path explosion* problem raised by symbolic pointers and loops are implemented inside of *SPF*. The *Searcher* is designed to assign more mutation energy to the promising seeds based on the weight value obtained by distance-based seed selection method. By doing this, the fuzzer will reach previously untouched more code areas as soon as possible in a given time budget.

In the experimental evaluation part, we address the following research questions:

- RQ1: Can BREACHER discover more deeper bugs?
- RQ2: Can distance-based seed selection method improve performance of path discovery?
- RQ3: How does each component contribute to path discovery?
- RQ4: Did BREACHER uncover unreported bugs in real-world binaries?

More specifically, RQ1 investigates the bug discovery ability of BREACHER on two benchmarks and compares the results with other vulnerability detection tools including fuzz testing and symbolic execution. For RQ2 and RQ3, we choose to use AFL's *unique paths* as performance evaluation criterion since it is a key factor to reflect the performance of a path-based program testing tool. RQ2 evaluates distance-based seed selection on several real-world programs to see whether our method can discover more unique paths than the state-of-the-art fuzzer: AFL. RQ3 evaluates the overall unique path discovery performance of BREACHER and investigates the performance contribution of each isolated component (i.e. *SPF* and *Searcher*). RQ4 benchmarks BREACHER on several real-world programs to check whether it can discover unreported crashes.

Evaluations were conducted on a server equipped with an Intel Xeon CPU E7-2850@2.0 GHz with 10 cores and 64 GB RAM, running Linux Ubuntu 14.04 LTS AMD64.

5.1 Vulnerability detection (RQ1)

We evaluated the bug discovery ability of our method with two different benchmarks. The first benchmark is a demo program which is named as *CommonMB*. The second benchmark is *LAVA*, which was released in 2016 to test different vulnerability discovery tools [21]. In the following sections, we are going to introduce the two benchmarks, and discuss the testing results of BREACHER as well as other off-the-shelf vulnerability discovery tools (AFL, VUzzer, KLEE, and S2E) in detail. AFL and VUzzer are coverage-based fuzzers. VUzzer leverages static analysis to extract comparison instructions, and then introduces these information into mutation to improve coverage [8]. By doing such, VUzzer is good at uncovering magic number related bugs. KLEE and S2E are both symbolic execution tools. However, KLEE works in user mode and can only handle compiled LLVM byte code. S2E can deal directly with binary executables on full software stack.

5.1.1 CommonMB: The *CommonMB* benchmark is a demo program which contains nine different memory error bugs. These bugs can be triggered only when feeding the program with specifically crafted input. There are four different kinds of functions in this benchmark, i.e. two compare-style functions, three math-style functions, two checksum-style functions, and two logic-style functions. The compare functions contain bugs that can only be triggered when the values of specific parts of the input equal to specific constant immediate numbers; the bugs in math functions can be triggered when the results of math operation on some specific parts of the input equal to specific constant immediate numbers; the checksum related bugs can only be triggered when the input data successfully goes through the checksum checking points; and the logic bugs utilise two simple logical games (maze and semi-sudoku) as the constraints for triggering the bugs, which means the bugs can only be triggered when the testing engine successfully solves the games. We released *CommonMB* benchmark on https://github.com/Epeius/CommonMB.git.

The bugs in *CommonMB* benchmark represent the common bug conditions in real-world programs. For example, compare-style bugs denote the bugs that depend on some specific/interesting values in the program; checksum-style bugs stands for the bugs whose inputs are well-formatted, such as PNG file. So evaluating a vulnerability detection tool on such a benchmark can reflect its ability of discovering bugs in real-world programs.

Table 2 shows the overall results of different vulnerability discovery tools as well as BREACHER with same test environment (10 cores and 12 h). We can see that all these tools have successfully triggered the two compare-style bugs (i.e. *cmp16* and *cmp32*). This is because there two functions are in the shallow surface of this benchmark and the conditions to trigger the bugs are simpler than the others.

Fuzz testing tool discovered few math-bugs than symbolic execution tools. AFL discovered only one bug in MATH (i.e. *add16*). It failed to uncover the bug that guarded by complex mathematic operations. VUzzer failed to detect any bugs in MATH. All tools that leverage symbolic execution successfully triggered all these three bugs since symbolic execution is good at solving such corner cases.

S2E provides function models for basic functions that may fork too many states, like `strcpy`, `strcat`, `crc16`, and `crc32`. Based on these models, S2E and BREACHER discovered the two bugs that related to checksum successfully (note that this does not mean S2E can break these checksum functions). Such bugs cannot be uncovered by KLEE.

Logic-style vulnerabilities are difficult for all tools to uncover because the number of states will be infinite in the worse case. For example, when solving a maze, the possible oscillation between two opposite steps (such as step forward and step backward) will stop the engine from finding new paths. With the help of our seed selection method, BREACHER assigned the seed file with the maximum average distance with more energy to mutate. Since this distance metric tries to maximise the memory coverage, BREACHER successfully triggered the bug after covering all the memory access to the maze array. However, our BREACHER failed to trigger the bug in *sudoku*. This is because different seed files of the *sudoku* have no significant differences (i.e. no more code/branch and memory coverage), which confuses our seed selection method.

Table 2 also presents the peak memory usage (PMU) of these tools in kB. From these data, we can see that BREACHER consumes more memory than other four tools, specifically, it brings memory overhead when compared with vanilla S2E. This is because we run both fuzzer and symbolic execution in the system. Since BREACHER's symbolic execution engine works on hybrid testing mode which only forks states for only uncovered branches of the seed from fuzzer, it brings less memory overhead (1.8% more).

5.1.2 LAVA benchmark: In 2016, Dolan-Gavitt *et al.* [21] developed a technique, namely LAVA, to automatically inject secure-related bugs into some Linux utilities for evaluating the bug-finding tools. These bugs are all hard-to-reach memory errors. In the paper of LAVA, the authors describe their results on the evaluation of coverage-based fuzz testing, an SAT-based approach on the benchmark. The LAVA benchmark has two corpus sets, i.e. *LAVA-1* and *LAVA-M*.

LAVA-1 injected 69 different bugs into the `file` program in Linux CoreUtils. Two types of buffer overflow vulnerabilities were injected, one is *Range* and the other one is *Knob-and-trigger (KT)*. The Range style bugs are triggered if the magic value is in some range and also check the value to determine how much to overflow.

Table 2 Evaluation results on *CommonMB* in detail

Tool		CMP		MATH			CHECKSUM		LOGIC		Total crashes (#)	PMU, kB
	version	cmp16	cmp32	add16	add32	complex	crc16	crc32	maze	sudoku		
AFL	2.52b	✓	✓	✓	✗	✗	✗	✗	✗	✗	3	4588.0
VUzzer	1.0	✓	✓	✗	✗	✗	✗	✗	✗	✗	3	25,702.4
KLEE (Random Searcher)	1.4.0	✓	✓	✓	✓	✓	✗	✗	✗	✗	5	108,236.8
S2E	2.0	✓	✓	✓	✓	✓	✓	✓	✗	✗	7	2,979,048.0
BREACHER	0.1	✓	✓	✓	✓	✓	✓	✓	✓	✗	8	3,031,373.2

Table 3 Evaluation results on *LAVA-1*

Tool	2^0 12 bugs	2^7 10 bugs	2^{14} 11 bugs	2^{21} 14 bugs	2^{28} 12 bugs	KT 10 bugs
FUZZER	0	0	1	11	9	2
SES	1	0	1	3	0	1
AFL	0	0	0	10	9	1
VUzzer	0	0	0	2	1	0
S2E	0	0	0	0	0	0
BREACHER$_E$	0	0	0	10	8	0
BREACHER$_C$	0	0	0	9	9	0
BREACHER$_J$	0	0	0	9	8	1
BREACHER	10	10	11	13	11	7

Table 4 Evaluation results on *LAVA-M*

Tool	`base64` (44 bugs)	`md5sum` (57 bugs)	`uniq` (28 bugs)	`who` (2136 bugs)	Total
FUZZER	7	2	7	0	16
SES	9	0	0	18	27
AFL	2	6	1	3	12
VUzzer	14	1*	24	103	142
S2E	1	0	0	2	4
BREACHER$_E$	1	6	1	4	12
BREACHER$_C$	2	4	1	4	11
BREACHER$_J$	1	4	1	3	9
BREACHER	37	29	28	203	297

In the KT bug, two bytes in the input are checked against a magic value to determine if the overflow will happen and another two bytes determine how much to overflow. Both the two types of bugs were designed to mirror real bug patterns which can be used to evaluate the ability of bug-finding tools. Compared to *LAVA-1*, which injected only one bug in the program, *LAVA-M* injected more than one bug into four different programs in CoreUtils that took file input: `base64`, `md5sum`, `uniq`, and `who`, so *LAVA-M* is a better benchmark to evaluate the vulnerability discovery tools that are designed to work for a long time on programs that may contain multiple bugs.

Note that since [21] does not make a statement about the details of about the tools and experimental setups they evaluated, we took their results directly into Tables 3 and 4. Also, because VUzzer runs only on 32-bit system, we re-executed VUzzer within a new environment running Linux Ubuntu 14.04 X86 with the same cores and memory as our experimental setup. Meanwhile, since VUzzer works on binary mode as well as S2E, *we discard to evaluate KLEE on LAVA benchmark*.

We also list BREACHER with different setups in Tables 3 and 4. Here BREACHER$*$ represents BREACHER with only *Searcher* disabled, where E, C, J denote *Euclidean distance*, *cosine similarity*, and *Jaccard index* metrics.

Table 3 summarised the results of bug finding evaluation on *LAVA-1* from the LAVA paper as well as some popular off-the-shelf tools. The maximum testing time for each bug was 5 h. From this table, we can see that the *FUZZER* and *SES* mentioned in the paper only found 23 bugs and 6 bugs, respectively, in total. AFL failed to trigger any bugs in smaller ranges (2^0, 2^7, and 2^{14}) but it outperformed VUzzer and S2E in larger ranges. VUzzer touched a total of three bugs which is less than AFL. This is because VUzzer's dynamic taint analysis (DTA) slowed down the testing speed but gained little in LAVA-1 since the conditions of triggering such bugs are not in strictly equality comparison form.

An interesting point is that S2E failed to trigger any bugs in LAVA-1. The reason is that because LAVA-1 is built on top of `file` program which contains many lookup tables by symbolic index, and such code will degrade symbolic execution to random-like fuzz testing (as shown in Fig. 3). Meanwhile, since S2E works on full system emulation mode and the execution speed is much less than fuzz testing in user mode like AFL, it failed to uncover bugs even in larger areas.

BREACHER discovered 62 bugs which was much more than the FUZZER and the SES tools separately. In particular, we triggered all the bugs in 2^7 and 2^{14} ranges. Also found most of the KT bugs (70%) which cannot be touched effectively by the FUZZER and SES tools. By leveraging LCSP algorithm, BREACHER's symbolic execution engine can go through the lookup tables to reach the bug points, and since the bugs' trigger conditions in LAVA-1 are easy for symbolic execution engine, BREACHER thus can uncover almost all the bugs in LAVA-1.

We can also derive from Table 3 that the gain of BREACHER in LAVA-1 mostly comes from *SPF* (BREACHER$*$ has the similar performance with AFL), especially in smaller ranges like 2^0, 2^7, and 2^{14}. This is because, even though the fuzzer engine can steer `file` program to the bug point, the trigger conditions are complex for BREACHER$*$ to overcome.

Table 4 describes the evaluation results on LAVA-M of the FUZZER and SES which are mentioned in the LAVA paper. We also listed other popular tools and BREACHER in this table. From this table we can see BREACHER outperformed other tools on this benchmark.

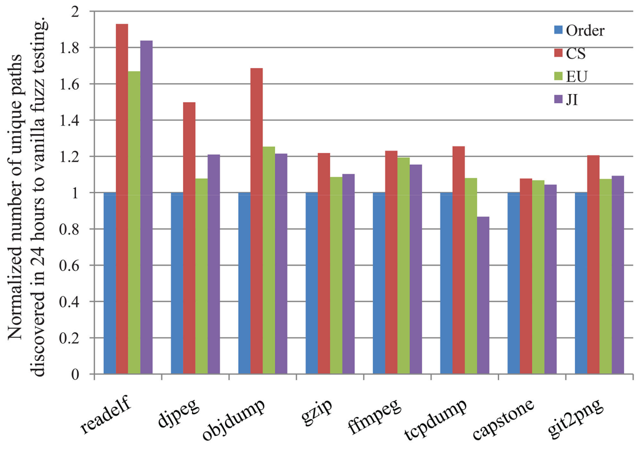
As mentioned in the LAVA paper, SES cannot find any bugs in `uniq` and `md5sum`. The reasons are the control flow is too unconstrained in `uniq` and SES failed to execute any code past the first instance of the hash function. VUzzer outperformed AFL in `base64`, `uniq`, and `who`, but it failed to trigger more bugs than AFL in `md5sum`. This is because it failed to get through the first crash to parse more of any input [8], whereas BREACHER successfully triggered 29 bugs in `md5sum`.

Similar to LAVA-1, the bug points in LAVA-M are also easy-to-reach. This is why BREACHER$*$ bring little performance improvement when compared with AFL. However, since *SPF* leverages symbolic execution to solve the bug conditions, more bugs are exposed.

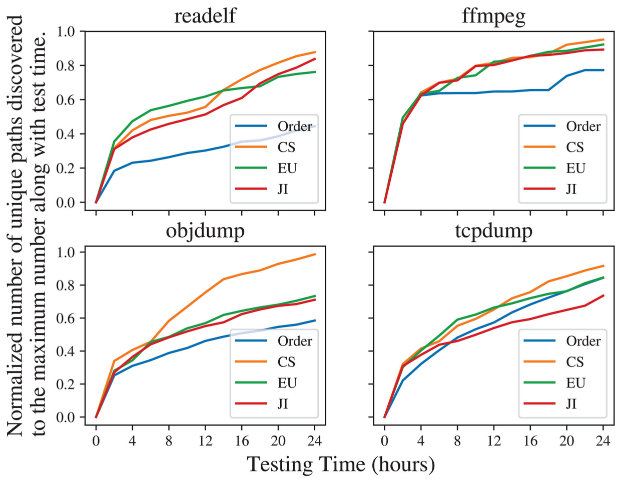
Since the role of the fuzzer (specifically, *Searcher*) in hybrid testing is to find as many code areas as possible, we devised more experiments to illustrate the coverage performance of distance-based seed selection method in Sections 5.2 and 5.3.

Table 5 Number of unique paths discovered for eight sample programs

Program	Order#	EU#	CS#	JI#
`readelf`	2753	4595	5314	5062
`djpeg`	2802	3020	4198	3390
`objdump`	1755	2200	2960	2133
`gzip`	1440	1564	1754	1588
`ffmpeg`	5022	5993	6181	5801
`tcpdump`	3399	3673	4267	2950
`capstone`	5626	6008	6066	5873
`gif2png`	912	981	1100	997



a



b

Fig. 9 *Path discovery results for different seed selection strategies*
(a) Normalised number of unique paths for these four selection strategies to vanilla fuzz testing (i.e. *Order*), ***(b)*** Path discovery details along with 24 h for four sample programs

5.2 Coverage performance (RQ2)

Our benchmark for coverage performance evaluation consists of eight real-world binary programs, and the input file formats of our benchmark cover a range of types such as executables, images, archives, and network packets.

To set the baseline of coverage performance, we utilised the state-of-the-art coverage-based fuzzer AFL [10] and configured it to run under *binary-testing* (i.e. option '-Q' is turned on) mode with one single work node. We collected and compared *the number of unique paths* found by AFL and our distance-based seed selection method. Then we select the most effective distance metric for subsection 5.3. All the evaluations lasted for 24 h.

The evaluation results were shown in Table 5. We have investigated the three distance measures mentioned before (i.e. EU, CS, and JI) as well as vanilla AFL (Order). From this table, we can see that by assigning more mutation energy to promising seeds, the fuzzer can touch more unique paths in average. More clearly, Fig. 9*a* shows the normalised unique paths to vanilla AFL for these eight programs based on the results in Table 5.

From Fig. 9*a*, we can see that both EU and CS metrics can outperform vanilla AFL on discovering unique paths for all programs in our benchmark. However, the average performance gain of EU is lowers than CS metric. Compared with the other two distance measures, JI is the most unstable strategy which discover more unique paths for some programs, like `readelf`, `djpeg`, but also brings performance overhead for some others, such as `tcpdump`.

An interesting point from Fig. 9*a* is that, for `capstone`, the performance gain of CS metric is not as significant as the other seven programs (found only 8% more paths than vanilla AFL). This is because, in our experiment, the input of `captone` was only plain texture file with some assembly code in it. Such kind of input is not as well formatted as other inputs like ELF, JPEG, CAP, and so on. So modifying any parts of the input may have same probability to trigger new behaviours which means each seed file in the queue will have nearly the same power to cover new code areas. This also demonstrates that our seed prioritisation method will gain more performance for well-formatted inputs.

To demonstrate the path discovery speed along with test time, we selected four representative programs, i.e., `readelf`, `ffmpeg`, `objdump`, and `tcpdump`, in our benchmark and collected the number of unique paths every 2 h (the number for each program is normalised to the maximum to make this figure more clear). The results are shown in Fig. 9*b*, where the *x*-axis indicates the test time in hours; while the *y*-axis shows the normalised unique number of paths triggered by each strategy. As shown in Fig. 9*b*, both CS and EU performed consistently better than orderly during all the 24 h. More specifically, EU performed better than CS in the first several hours, and then CS outperformed EU in the following test time. While JI performed well in `readelf`, `ffmpeg`, and `objdump`, but it failed to improve the performance in `tcpdump` after testing for 8 h.

Especially for `ffmpeg`, we can see that in the first 4 h, all of these four selection strategies achieved the same performance on discovering unique paths. However, vanilla AFL failed to trigger more unique paths during 4–18 h before it started to find new paths again. This is because the mutation areas of seeds that processed during 4–18 h are overlapped with each other (as shown in Fig. 7), which will not contribute any new paths. However, our distance-based seed selection strategy can consistently trigger new paths during 24 h as shown in this figure.

Based on the results, we can obtain that our distance-based seed selection strategy (especially CS metric) can achieve higher performance than vanilla AFL. So we selected CS metric as our selection strategy in the following evaluation section.

5.3 Improvement detail for each component (RQ3)

In order to evaluate the contribution of each component on finding unique paths, we conducted four experiments [i.e. vanilla AFL fuzz testing, symbolic execution assisted hybrid testing (*SEHT*), SEHT with *SPF*, SEHT with *SPF&Searcher*] and compared their results in 24 h.

In our experiments, since Driller [9] does not support real-world binary programs very well now (many library functions and system calls are not modelled), we reimplemented the mechanism of Driller within S2E to collect the results for SEHT. Based on the results of Section 5.2, our searcher employs *cosine similarity* as the distance metric.

Fig. 10 summarises the results of how each component contributes to the performance improvement. Compared with vanilla fuzz testing, SEHT can discover 11.88% more unique paths in average. For example, this improvement reaches the highest figure of 42.79% for `readelf`. However, the performance gain is lower than 20% for the other seven programs. Specifically, SEHT triggered only 1.32% more unique paths than vanilla fuzz testing for `gif2png`. This is because the symbolic execution engine does not support to handle float number operation. After employing *SPF* which handles symbolic pointers and symbolic loops, 7.22% more

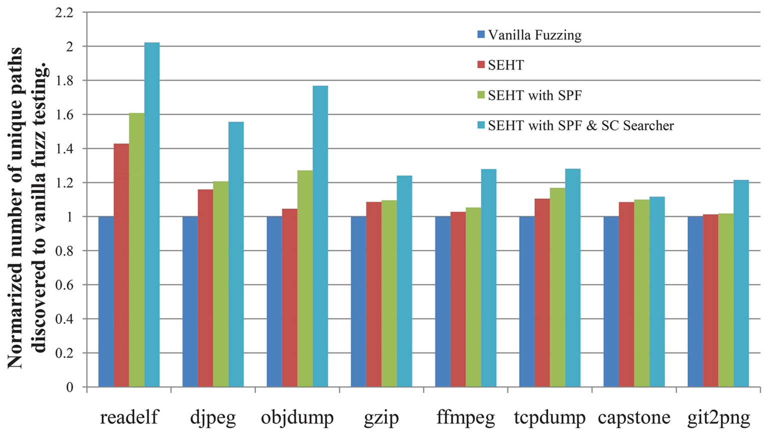


Fig. 10 *Normalised number of unique paths discovered for each component to vanilla fuzz testing*

Table 6 Performance of our method versus AFL and VUzzer on unknown crashes

Program	Input Type	BREACHER		AFL		VUzzer	
		Crashes (#)	Executed (#)	Crashes (#)	Executed (#)	Crashes (#)	Executed (#)
`elfparser`	ELF	60	1.1M	48	1.0M	33	19.0k
`distorm`	ELF	13	16.2M	0	15.7M	3	93.1k
`mp3gain*`	MPEG	43	11.3M	41	11.7M	46	79.3k
`madplay*`	WAV	55	15.9M	54	14.3M	52	88.7k
`optipng`	PNG	49	9.5M	15	10.1M	32	54.1k
`tstat`	PCAP	183	6.4M	23	6.2M	26	48.5k
total		403		181		192	

Star symbol means that Breacher failed to uncover more paths in these two programs since our symbolic execution engine does not support float point number operations.

unique paths touched by SEHT with *SPF* than SEHT in average. We can see from this figure that the performance of SEHT is highly improved by *SPF* for `readelf` and `objdump` (18.02 and 22.62%, respectively).

This improvement can be continually augmented by cooperating with CS searcher (SEHT with *SPF* and CS *Searcher*). For example, the number of unique paths discovered by SEHT is increased by 49.57% for `objdump`. In average, the performance of SEHT is improved by 24.44% after introducing *SPF* and CS *Searcher*. From an overall viewpoint, BREACHER can discover 43.49% more unique paths than vanilla AFL fuzz testing for our benchmark. This improvement is because that seeds files with longest distance from already-explored spaces have higher likelihood to trigger more fresh branches/paths, so solving such seed files earlier by symbolic execution can find more fresh seed files than other files.

Above all, we can obtain that *Searcher* contributes larger proportion of contribution to unique path discovery than *SPF*. However, this does not mean the contribution of *SPF* is negligible. By integrating *SPF* component (i.e. *LCSP* and *SLB*), the symbolic execution engine can dive into deeper code areas to solve *complex branch conditions* to help fuzzer find more fresh paths and discover more hard-to-reach vulnerabilities (as shown in Section 5.1).

5.4 Unknown crashes discovered in real-world binaries (RQ4)

We selected several real-world programs to evaluate the ability of unknown bug discovery of our system. The input types of this dataset cover ELF binary, multi-media, image, and packet capture. In order to demonstrate the efficiency of our system, we also compared our results with AFL and VUzzer. We ran each program under the same testing environment with one fuzzing node for 24 h.

Table 6 shows the results of our testing. From the table we can see that during 24 h, BREACHER triggered 403 unique crashes in the dataset which outperformed vanilla AFL (181 unique crashes) and VUzzer (192 unique crashes). We can also derive from this table that our method gains little for `mp3gain` and `madplay`. This is because our symbolic execution engine does not support the float number operation when handing MPEG/WAV format (e.g. all the writing operation to `XMM` registers will be concretised). This concretisation lost some interesting paths and made less contribution to bug finding (only two more bugs for `mp3gain` and one more bug for `madplay`). This also explains why VUzzer found more bugs in `mp3gain` than BREACHER. It is interesting that AFL detected 15 more bugs in `elfparser` than VUzzer. This is because the fork server method leveraged in AFL enables the fuzzer can execute 51.6x more test cases than VUzzer in the same time, which can also increase the probability of finding bugs.

6 Limitations and discussion

Our method is built on top of coverage-based fuzz testing and dynamic symbolic execution, where we have introduced distance-based seed search strategy, SLB, and lazy symbolic pointer. While an improvement, there are still some drawbacks to our method. This section discusses these limitations and takes a future look at the vulnerability discovery.

6.1 Limitations

Distance measurement: Our seed selection strategy leverages three well-known distance measures, i.e. Euclidean distance, cosine similarity, and Jaccard index. Also, we have evaluated these three measures and compared the results with no search strategy. In the future, we hope to investigate other distance metrics (e.g. hamming distance, N-gram distance etc.) to find a better measurement for different execution paths (or different seed inputs).

Plain input format: Programs that accept input with no specific format cannot gain performance improvement from our distance-based seed selection method. As shown in Fig. 9*a*, all of these three distance-based selection strategies failed to trigger more new behaviours for `capstone` which accepts the plain texture file as input.

Float point operation: The dynamic symbolic execution engine we depend on will concretise symbolic write operations to XMM registers, which will lose some interesting paths when handling float point arithmetic operations. For example, the *Floating-point*

Number benchmark in [37] makes an obstacle for KLEE to solve. This problem also happens for most of the image processing programs. So the dynamic symbolic execution engine should be upgraded to support float point arithmetic operations to handle these types of programs.

6.2 Future work on vulnerability detection

This section will briefly propose some possible future research areas in vulnerability detection.

6.2.1 Binary transformations: Dynamic symbolic execution still faces scalability problem when considering the size and complexity of modern software. Compiler optimisations can have a large impact on dynamic symbolic execution's effectiveness. In 2013, J. Wagner *et al.* proposed a new compiler option -OVERIFY to generate code optimised specifically for verification tools. Their experiments' results show that -Overify can reduce verification time by up to 95x for GNU Coreutils [38]. As discussed in [39], LLVM compiler's -O0 flag can contribute different paths from flag -O2 (which contributes 1024 and 2 paths, respectively, for its sample code). Cadar claims that one should treat program transformations as first-class ingredients of scalable symbolic execution, alongside widely-accepted aspects such as search heuristics and constraint solving optimisations [39].

With this insight, we propose that in the future, binary executables should be pre-processed to transform *testing-expensive* code structures to *testing-cheap* ones. For example, transforming a long string comparison instruction to several isolated byte comparison instructions will improve the performance of fuzz testing. Similar to [40], the binary pre-processing stage should recognise which code areas are the testing 'hot spot' and remove/ transform them to avoiding getting stuck in such areas. These transformations can either be semantic-preserving or semantic-altering transformations. The key research problem is how to perform these transformations on binary level, since most high level program information is lost during compilation. On possible solution to achieve such transformations is to lift binary code into intermediate expression such as LLVM byte code, then perform more analysis to recover program information such as control flow graph, data dependency, and control dependency.

6.2.2 Finding bugs with machine learning: Many research papers treat the vulnerability detection with dynamic symbolic execution and fuzz testing as a search problem. Since the search space of modern software can be vast, exhaustive exploration of this space is currently impossible. Reducing the search space may lead to better coverage. However, this may miss interesting sub-spaces where contain vulnerabilities. In order to find deeper bugs when reducing the search space, one has to locate *secure-sensitive code areas* based on static analysis, and then uses search heuristics to guide the program exercise these areas. Since each type of vulnerability has its own unique characteristic [41–43], some researchers have attempted to use machine learning to automatically extract such characteristics in source code, and then predict potential vulnerabilities [44, 45].

In 2016, Grieco *et al.* proposed a binary software vulnerability predict tool VDISCOVER as well as a public dataset that collects raw analysed data [46]. They 'managed to predict with reasonable accuracy which programs contained dangerous memory corruptions' [46]. Based on such work, we propose that in the future, machine learning could be ported to binary software vulnerability detection by cooperating with guided testing techniques. Since machine learning can raise many false positives, one can leverage guided dynamic symbolic execution to mitigate the false positives and verify the existence of potential vulnerabilities.

7 Related work

We have presented the major advantages of our method in the previous sections and compared our system with some state-of-the-art vulnerability discovery tools. In this section, we present the techniques that related to our method.

Similarity distance in regression testing: Similarity based algorithms have previously been leveraged in regression test case prioritisation [34, 47, 48]. Test case prioritisation is a hot research topic in regression testing research, which tries to optimum mutation schedule based on a specific prioritisation criterion. Rothermel *et al.* proposed fine-grained prioritisation strategy based on the instruction coverage and branch coverage [49]. Then Elbaum *et al.* concentrated on function level coverage and they proved that this kind of coarse-grained instrumentation which can reduce the execution overhead but will lose some prioritisation performance [50]. Krishna *et al.* utilised Levenshtein distance as the criterion of prioritisation [51]. Rather than using an ordered branch sequence to present the path in [34], we represented the execution path by using the bitmap in AFL, which is more practical and efficiency.

Taint analysis based fuzz testing: Taint analysis based fuzz testing uses DTA to locate regions of seed input that affect the execution path. BuzzFuzz uses DTA to automatically locate regions of original seed input files that influence values used at key program attack points, and then automatically generates new fuzzed test input files by fuzzing these identified regions of the original seed input files [5]. TaintScope is a directed fuzzing tool working at X86 binary level. Based on fine-grained DTA, TaintScope identifies which bytes in a well-formed input are used in security-sensitive operations (e.g. invoking system/library calls) and then focuses on modifying such bytes. TaintScope is also capable of bypassing checksums via control flow alteration [52]. Dowser is a guided fuzzer that combines static analysis, DTA, and symbolic execution to find buffer overflow vulnerabilities deep in a program's logic, and it ranks pointer dereference instructions according to their complexity, and then uses symbolic execution to zoom in on the most interesting operation [53].

Dealing with symbolic pointers: The symbolic pointer problem occurs when the program dereferences a symbolic address. Previous work on symbolic execution deals with this problem by leveraging the full symbolic memory model [24–27]. Under this model, if an instruction reads/writes to a symbolic address, then each possible value of this address will fork a corresponding state so that all possible paths can be exercised. As previously discussed, forking state for each possible value will quickly cause path explosion. In 2014, Mayhem [31] introduced a partial memory model to ease the scalability problem inherit in the full symbolic memory model. In the partial memory model, states are only forked when reading a symbolic pointer. A write operation causes the symbolic pointer to be concretised. This can reduce the number of forked states, but a high number of read operations may still cause path explosion. Our LCSP method takes advantages of S2E's lazy forking to postpone the path explosion problem to a later moment. LCSP can also improve code coverage by solving the pending states from lazy forking on demand. Recently, MEMSIGHT [54] introduces a new approach to symbolic memory that reduces the need for concretisation but still can explore more states. It leverages stage merging [55] to compact symbolic pointer read operations. Some optimisations are introduced to deal with performance issues in MEMSIGHT, e.g. it constraints the range of symbolic pointer to a certain interval by leveraging SMT solver, which we also used in BREACHER. It also proposed a memory-wise *paged interval tree* to enable better memory space usage. The experimental results compared with Angr show MEMSIGHT enables the exploration of states unreachable by previous techniques. Since we are dealing with the symbolic pointer problem in hybrid testing of real-world binaries, which currently Angr cannot support very well now (e.g. complex library function model problem), we will investigate more about MEMSIGHT in the near future to adopt its advantages in hybrid testing.

Hybrid testing method: As mentioned before, our method is not the first tool to combine fuzz testing and symbolic execution. Hybrid fuzz testing uses symbolic execution to discover frontier nodes that represent unique paths in the program [13]. After collecting as many frontier nodes as possible under a user-specifiable resource constraint, it transits to fuzz the program with random inputs. This tool focuses on binaries but only performs the one-time transition between symbolic execution and fuzz testing. Hybrid concolic testing implements multiple transitions between symbolic execution and fuzz testing [12]. However, because it is built on top of CUTE, a source code oriented testing tool, so hybrid concolic testing still cannot be deployed on binary testing directly [56]. Driller is an up-to-date hybrid testing tool that leverages fuzz testing and concolic execution in a complementary manner to find deeper bugs [9]. It is more practice when compared with previous hybrid tools. Some other tools try to make full use of symbolic execution to maximise the code coverage, they collect symbolic constraints placed on each input and then negating these constraints to generate a new test case that will take another uncovered path, such as SAGE [11], Dowser [53], FuzzWin [57] and so on. However, as these tools execute each input in the symbolic mode which determines that they have to face the path explosion problem.

8 Conclusion

In this paper, we focused on improving the performance of hybrid testing method built on coverage-based fuzz testing and dynamic symbolic execution. We proposed a novel method, lazy concretisation, to deal with symbolic pointers. We found that this method mitigates the path explosion problem and improves code coverage. We also introduced the loop bucket optimisation in order to avoid generating too many states in symbolic loops. In order to deal with the large size of the seed queue in hybrid testing, we presented a distance-based seed selection method to achieve more coverage when testing time is limited. This criteria of selection method is built on top of runtime information (i.e. path and memory information). The evaluation of BREACHER on several benchmarks demonstrates that our method can discover more unique paths than vanilla fuzz testing and finds more bugs compared with other off-the-shelf vulnerability analysis tools.

9 References

[1] Duran, J.W., Ntafos, S.C.: 'An evaluation of random testing', *IEEE Trans. Softw. Eng.*, 1984, **SE-10**, (4), pp. 438–444

[2] Miller, B.P., Fredriksen, L., So, B.: 'An empirical study of the reliability of unix utilities'. Proc. of the Workshop of Parallel and Distributed Debugging, Santa Cruz, CA, USA, 1990, pp. ix–xxi

[3] Sutton, M., Greene, A., Amini, P.: '*Fuzzing: brute force vulnerability discovery*' (Pearson Education, New York, NY, USA, 2007)

[4] Cadar, C., Godefroid, P., Khurshid, S., *et al.*: 'Symbolic execution for software testing in practice: preliminary assessment'. Proc. of the 33rd Int. Conf. on Software Engineering, Waikiki, HI, USA, 2011, pp. 1066–1071

[5] Ganesh, V., Leek, T., Rinard, M.: 'Taint-based directed whitebox fuzzing'. Proc. of the 31st Int. Conf. on Software Engineering, Vancouver, Canada, 2009, pp. 474–484

[6] Godefroid, P., de Halleux, P., Nori, A.V., *et al.*: 'Automating software testing using program analysis', *IEEE Softw.*, 2008, **25**, (5), pp. 30–37

[7] Neystadt, J.: '*Automated penetration testing with white-box fuzzing*' (MSDN Library, 2008)

[8] Rawat, S., Jain, V., Kumar, A., *et al.*: 'Vuzzer: application-aware evolutionary fuzzing'. Proc. of the Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2017

[9] Stephens, N., Grosen, J., Salls, C., *et al.*: 'Driller: augmenting fuzzing through selective symbolic execution'. Proc. of the Network and Distributed System Security Symp., San Diego, CA, USA, 2016

[10] Zalewski, M.: 'American fuzzy lop'. Available at http://lcamtuf.coredump.cx/afl/

[11] Godefroid, P., Levin, M.Y., Molnar, D.: 'Sage: whitebox fuzzing for security testing', *Queue*, 2012, **10**, (1), p. 20

[12] Majumdar, R., Sen, K.: 'Hybrid concolic testing'. 29th Int. Conf. on Software Engineering. ICSE 2007, Minneapolis, MN, USA, 2007, pp. 416–426

[13] Pak, B.S.: 'Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution'. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2012

[14] Yeh, C.-C., Chung, H., Huang, S.-K.: 'Craxfuzz: target-aware symbolic fuzz testing'. 2015 IEEE 39th Annual Computer Software and Applications Conf. (COMPSAC), Taichuang, Taiwan, 2015, vol. 2, pp. 460–471

[15] Shoshitaishvili, Y., Wang, R., Hauser, C., *et al.*: 'Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware'. Network and Distributed System Security, San Diego, CA, USA, 2015

[16] DARPA: 'Cyber grand challenge'. Available at http://cybergrandchallenge.com

[17] Baldoni, R., Coppa, E., D'Elia, D.C., *et al.*: 'A survey of symbolic execution techniques'. arXiv preprint arXiv:1610.00502, 2016

[18] Boonstoppel, P., Cadar, C., Engler, D.: 'Rwset: attacking path explosion in constraint-based test generation'. Proc. of the Theory and Practice of Software, 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Berlin, Heidelberg, 2008, pp. 351–366

[19] Schwartz, E.J., Avgerinos, T., Brumley, D.: 'All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)'. 2010 IEEE Symp. on Security and Privacy (SP), Berkeley, CA, USA, 2010, pp. 317–331

[20] Chipounov, V., Kuznetsov, V., Candea, G.: 'S2e: A platform for in-vivo multi-path analysis of software systems'. ASPLOS, Newport Beach, CA, USA, 2011

[21] Dolan-Gavitt, B., Hulin, P., Kirda, E., *et al.*: 'Lava: large-scale automated vulnerability addition'. 2016 IEEE Symp. on Security and Privacy (SP), San Jose, CA, USA, 2016, pp. 110–121

[22] King, J.C.: 'Symbolic execution and program testing', *Commun. ACM*, 1976, **19**, (7), pp. 385–394

[23] Cadar, C., Sen, K.: 'Symbolic execution for software testing: three decades later', *Commun. ACM*, 2013, **56**, (2), pp. 82–90

[24] Brumley, D., Jager, I., Avgerinos, T., *et al.*: 'Bap: A binary analysis platform'. Int. Conf. on Computer Aided Verification, Snowbird, UT, USA, 2011, pp. 463–469

[25] Song, D., Brumley, D., Yin, H., *et al.*: 'Bitblaze: A new approach to computer security via binary analysis'. Int. Conf. on Information Systems Security, Hyderabad, India, 2008, pp. 1–25

[26] Thakur, A., Lim, J., Lal, A., *et al.*: 'Directed proof generation for machine code'. Int. Conf. on Computer Aided Verification, Edinburgh, UK, 2010, pp. 288–305

[27] Trtık, M., Strejcek, J.: 'Symbolic memory with pointers'. Automated Technology for Verification and Analysis: 12th Int. Symp., ATVA 2014, Sydney, Australia, 3–7 November 2014, Proc., Sydney, Australia, 2014, vol. 8837, p. 380

[28] Cadar, C., Dunbar, D., Engler, D.R., *et al.*: 'Klee: unassisted and automatic generation of high-coverage tests for complex systems programs'. Operating Systems Design and Implementation, San Diego, CA, USA, 2008, vol. 8, pp. 209–224

[29] Cadar, C., Ganesh, V., Pawlowski, P., *et al.*: 'Exe: A system for automatically generating inputs of death using symbolic execution'. Proc. of the ACM Conf. on Computer and Communications Security, Alexandria, VA, USA, 2006

[30] Avgerinos, A.: 'Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs'. PhD thesis, Carnegie Mellon University, CMU, 2014

[31] Cha, S.K., Avgerinos, T., Rebert, A., *et al.*: 'Unleashing mayhem on binary code'. 2012 IEEE Symp. on Security and Privacy (SP), San Francisco, CA, USA, 2012, pp. 380–394

[32] Avgerinos, T., Cha, S.K., Rebert, A., *et al.*: 'Automatic exploit generation', *Commun. ACM*, 2014, **57**, (2), pp. 74–84

[33] Kim, S.Y., Lee, S., Yun, I., *et al.*: 'Cab-fuzz: practical concolic testing techniques for cots operating systems'. 2017 USENIX Annual Technical Conf. (USENIX ATC 17), Santa Clara, CA, 2017, pp. 689–701

[34] Wang, R., Jiang, S., Chen, D.: 'Similarity-based regression test case prioritization'. Software Engineering and Knowledge Engineering, Pittsburgh, PA, USA, 2015, pp. 358–363

[35] Bellard, F.: 'QEMU, a fast and portable dynamic translator'. USENIX Annual Technical Conf., FREENIX Track, Anaheim, CA, USA, 2005, pp. 41–46

[36] Lattner, C., Adve, V.: 'Llvm: A compilation framework for lifelong program analysis & transformation'. Proc. of the Int. Symp. on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, Washington, DC, USA, 2004, p. 75

[37] Xu, H., Zhao, Z., Zhou, Y., *et al.*: 'On benchmarking the capability of symbolic execution tools with logic bombs'. arXiv preprint arXiv:1712.01674, 2017

[38] Wagner, J., Kuznetsov, V., Candea, G.: 'Overify: optimizing programs for fast verification'. 14th Workshop on Hot Topics in Operating Systems (HotOS XIV), Santa Ana Pueblo, NM, USA, 2013, number EPFL-CONF-186012

[39] Cadar, C.: 'Targeted program transformations for symbolic execution'. Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, New York, NY, USA, 2015, pp. 906–909

[40] Wagner, J., Kuznetsov, V., Candea, G., *et al.*: 'High system-code security with low overhead'. 2015 IEEE Symp. on Security and Privacy (SP), San Jose, CA, USA, 2015, pp. 866–879

[41] Bishop, M., Engle, S., Howard, D., *et al.*: 'A taxonomy of buffer overflow characteristics', *IEEE Trans. Dependable Secur. Comput.*, 2012, **9**, (3), pp. 305–317

[42] Wang, T., Wei, T., Lin, Z., *et al.*: 'Intscope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution'. Network and Distributed System Security, San Diego, CA, USA, 2009

[43] Wang, Y., Gu, D., Xu, J., *et al.*: 'Ricb: integer overflow vulnerability dynamic analysis via buffer overflow'. Int. Conf. on Forensics in Telecommunications, Information, and Multimedia, Shanghai, China, 2010, pp. 99–109

[44] Perl, H., Dechand, S., Smith, M., *et al.*: 'Vccfinder: finding potential vulnerabilities in open-source projects to assist code audits'. Proc. of the 22Nd ACM SIGSAC Conf. on Computer and Communications Security, CCS'15, New York, NY, USA, 2015, pp. 426–437

[45] Yamaguchi, F., Lindner, F., Rieck, K.: 'Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning'. Proc. of the 5th USENIX Conf. on Offensive Technologies, WOOT'11, Berkeley, CA, USA, 2011, pp. 13–13

[46] Grieco, G., Grinblat, G.L., Uzal, L., *et al.*: 'Toward large-scale vulnerability discovery using machine learning'. Proc. of the Sixth ACM Conf. on Data and Application Security and Privacy, CODASPY '16, New York, NY, USA, 2016, pp. 85–96

[47] Jones, J.A., Harrold, M.J.: 'Test-suite reduction and prioritization for modified condition/decision coverage', *IEEE Trans. Softw. Eng.*, 2003, **29**, (3), pp. 195–209

[48] Zhang, D., Liu, D., Lei, Y., *et al.*: 'SimFuzz: test case similarity directed deep fuzzing', *J. Syst. Softw.*, 2012, **85**, (1), pp. 102–111

[49] Rothermel, G., Untch, R.H., Chu, C., *et al.*: 'Prioritizing test cases for regression testing', *IEEE Trans. Softw. Eng.*, 2001, **27**, (10), pp. 929–948

[50] Elbaum, S., Malishevsky, A., Rothermel, G.: 'Incorporating varying test costs and fault severities into test case prioritization'. Proc. of the 23rd Int. Conf. on Software Engineering, Washington, DC, USA, 2001, pp. 329–338

[51] Krishnamoorthi, R., Sahaaya Arul Mary, S.A.: 'Factor oriented requirement coverage based system test case prioritization of new and regression test cases', *Inf. Softw. Technol.*, 2009, **51**, (4), pp. 799–808

[52] Wang, T., Wei, T., Gu, G., *et al.*: 'Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection'. 2010 IEEE Symp. on Security and Privacy (SP), Berkeley, CA, USA, 2010, pp. 497–512

[53] Haller, I., Slowinska, A., Neugschwandtner, M., *et al.*: 'Dowsing for overflows: A guided fuzzer to find buffer boundary violations'. USENIX Security, Washington, DC, USA, 2013, pp. 49–64

[54] Coppa, E., D'Elia, D.C., Demetrescu, C.: 'Rethinking pointer reasoning in symbolic execution'. Proc. of the 32Nd IEEE/ACM Int. Conf. on Automated Software Engineering, ASE 2017, Piscataway, NJ, USA, 2017, pp. 613–618

[55] Avgerinos, T., Rebert, A., Cha, S.K., *et al.*: 'Enhancing symbolic execution with veritesting'. Proc. of the 36th Int. Conf. on Software Engineering, ICSE 2014, New York, NY, USA, 2014, pp. 1083–1094

[56] Sen, K., Marinov, D., Agha, G.: 'Cute: a concolic unit testing engine for c'. ACM SIGSOFT Software Engineering Notes, New York, NY, USA, 2005, vol. 30, pp. 263–272

[57] Lecomte, S.: 'Fuzzwin: a new concolic tool using a pin ir+z3'. Available at https://github.com/piscou/FuzzWin