EPfuzzer: Improving Hybrid Fuzzing with Hardest-to-reach Branch Prioritization

Yunchao Wang, Zehui Wu, Qiang Wei*, and Qingxian Wang
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China
[e-mail: 92wyunchao@gmail.com, wuzehui2010@foxmail.com, prof_weiqiang@163.com, wangqingxian2015@163.com]
*Corresponding author: Qiang Wei

Received January 22, 2020; revised July 14, 2020; accepted August 23, 2020; published September 30, 2020

Abstract

Hybrid fuzzing which combines fuzzing and concolic execution, has proved its ability to achieve higher code coverage and therefore find more bugs. However, current hybrid fuzzers usually suffer from inefficiency and poor scalability when applied to complex, real-world program testing. We observed that the performance bottleneck is the inefficient cooperation between the fuzzer and concolic executor and the slow symbolic emulation. In this paper, we propose a novel solution named EPfuzzer to improve hybrid fuzzing. EPfuzzer implements two key ideas: 1) only the hardest-to-reach branch will be prioritized for concolic execution to avoid generating uninteresting inputs; and 2) only input bytes relevant to the target branch to be flipped will be symbolized to reduce the overhead of the symbolic emulation. With these optimizations, EPfuzzer can be efficiently targeted to the hardest-to-reach branch. We evaluated EPfuzzer with three sets of programs: five real-world applications and two popular benchmarks (LAVA-M and the Google Fuzzer Test Suite). The evaluation results showed that EPfuzzer was much more efficient and scalable than the state-of-the-art concolic execution engine (QSYM). EPfuzzer was able to find more bugs and achieve better code coverage. In addition, we discovered seven previously unknown security bugs in five real-world programs and reported them to the vendors.

Keywords: Bug detection, concolic execution, hybrid fuzzing, software security

This work was supported by National Key R&D Program of China under Grant 2019QY0501 and Soft Science Research Program of Henan Province under Grant 192102210128.

1. Introduction

Software vulnerabilities [1, 2, 3] are a main threat to program security. As a result, researchers from industry and academia have come up with many automatic methods to find vulnerabilities in programs. Fuzzing [4, 5, 6] and concolic execution [7, 8, 9] are two popular vulnerability detection techniques. On the one hand, fuzzing can quickly cover many branches with simple or loose constraint conditions (a large range of satisfying value space), such as the two types shown in **Fig. 1a**. However, due to the randomness of mutation, it is difficult for fuzzer to cover branches guarded by complex or tight constraints (very few range of satisfying value space), such as the three types shown in **Fig. 1b**. For example, it is hard for the state-of-the-art American Fuzzy Lop (AFL) [4] to generate inputs that meet these branch conditions and find bugs hidden behind them. Although some heuristic strategies [10, 11, 12, 13] have been proposed to alleviate this limitation, they are only applicable for comparatively simple situation like direct multibyte check (e.g., ③ in **Fig. 1b**), but not for a more complex indirect multibyte check (e.g., ④ in **Fig. 1b**) and nested condition check (e.g., ⑤ in **Fig. 1b**). On the other hand, concolic execution is very good at solving these complex branch constraints, but it is not scalable to real-world applications due to high overhead (symbolic emulation and constraint solving are very time-consuming) and path explosion problems.

```
①. if(x == 0x41)  // Single byte check
②. if(x > 1)  // Loose condition check
```

(a) Simple branch condition for fuzzer

```
③. if(x == 0xdeadbeef)  // Direct multibyte check
④. if(2x+1 == 0xdeadbeef)  // Indirect multibyte check
⑤. if(x+y < 10)
       if(x > 5)  // Nested condition check
```

(b) Complex branch condition for fuzzer

Fig. 1. Some common branch condition checks in real-world programs.

To make full use of the strengths of fuzzing and concolic execution and mitigate their weaknesses, a new method named hybrid fuzzing [14, 15, 16, 17] was proposed with the aim of maximizing code coverage of the fuzzer. This method lets the high-speed fuzzing run as much as possible for path traversal, and the expensive concolic execution is only used to assist in solving some complex branch constraints to generate test cases that can cover specific branches, thereby helping fuzzing to further improve code coverage and find deeper code bugs. Driller [15] is a representative hybrid fuzzer and has proved that techniques combining fuzzing and concolic execution can find more bugs in the DARPA Cyber Grand Challenge (CGC) binaries than techniques using either method alone. DigFuzz [16] designs a novel Monte Carlo based probabilistic path prioritization model to quantify each path's difficulty and prioritize them for concolic execution, which further helps to improve the efficiency of hybrid fuzzing. Unfortunately, Driller and DigFuzz are both not scalable to nontrivial, real-world applications, because its symbolic execution engine, angr [9], is very slow and many external system calls cannot be supported. QSYM [17] is a recently proposed and more practical concolic execution engine tailored for hybrid fuzzing. It integrates symbolic emulation with native execution by dynamic binary translation instead of depending on slower intermediate representation. The concolic executor QSYM and fuzzer run in parallel. QSYM reruns the generated seed inputs from the fuzzer and then flips each condition branch in turn and solves the corresponding path constraint to generate a new input to the fuzzer. However, we found that this strategy has two problems: 1) QSYM flips all conditional branches in the execution path indiscriminately, resulting in most of the generated input being uninteresting to the fuzzer and discarded. Consequently, the concolic executor spends considerable time doing much useless work; 2) QSYM sees the all user input as symbolic bytes, thus increasing the burden of symbolic emulation and constraint solving. Consequently, a single concolic execution takes too much time. These two issues greatly reduce the overall performance of the hybrid fuzzer.

To overcome the above limitations and further improve the scalability of hybrid fuzzing, we propose a hardest-to-reach branch prioritization strategy. First, instead of solving all branches blindly, we only choose branches that are really hard to reach for the fuzzer for concolic execution. Second, we only symbolize bytes that affect target branch to be flipped, thereby reducing the number of symbolic instructions which need to be emulated. We implement a prototype tool named EPfuzzer and evaluate it with three sets of programs: five real-world applications and two benchmarks (LAVA-M [18] and Fuzzer Test Suite [19]). The experimental results show that our method can achieve higher code coverage and discover more bugs than QSYM. In summary, this paper makes the following contributions:

1) We propose a hardest-to-reach branch prioritization strategy to improve the performance of hybrid fuzzing. Only the hardest-to-reach branches are solved using concolic execution to generate new inputs to avoid generating many uninteresting inputs. Additionally, only specific bytes are symbolized to reduce the overhead of symbolic emulation.

2) We implement EPfuzzer and evaluate it with state-of-the-art fuzzers (e.g., QSYM and AFL) on two benchmarks and five real-world applications. EPfuzzer outperforms in terms of both code coverage and bug discovery.

3) We found seven previously unknown security bugs and reported them to the vendor.

2. Background and Motivation

This section illustrates the related research background and the motivation of our study.

2.1 Background

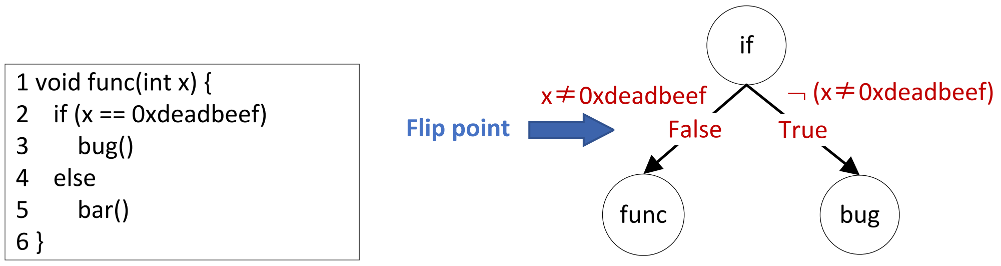
Greybox fuzzing and concolic execution are two well-known vulnerability discovery methods and in the meantime they are aslo two key componets of a hybrid fuzzing system. Therefore, we first make a brief introduction to them before illustrating our motivation.

(1) Greybox fuzzing

The greybox fuzzers such as AFL have achieved a great success and found a large number of security vulnerabilities. It leverages lightweight instrumentation and genetic algorithms to discover test cases that likely trigger new branch in the target program. To further improve the code coverage, current greybox fuzzers adopt many heuristic strategies. AFLFast [20] and FairFuzz [21] prioritize seeds that trigger the low-frequency paths or rare parts of the program. Collafl [22] proposes three new seed selection policies to drive the fuzzer directly toward non-explored paths. These fuzzers can achieve higher code coverage to some extent, but they cannot solve complex constraints. For example, it is hard for fuzzers mentioned above to generate an input which can satisfy the constraint in a conditional branch such as "if(x==0xdeadbeef)", because it has to guess a single value out of a large space (2^{32}).

(2) Concolic execution

Concolic execution (also known as dynamic symbolic execution) such as s2e [8], angr [9] is a popular path-exploring technology. It treats the user input as symbolic values along a concrete execution path. Whenever the engine encounters a conditional branch where the branch predicate is symbolic, the branch will be flipped and a constraint solver is used to create a new input by solving collected path constraints. Take the code shown in Fig. 2 as an example, suppose that the initial input is x=0x00000000, then the false branch will be explored. To explore the true branch, the concolic executor will flip the false branch, that is, obtain new path constraints by negating the current branch constraint and then ask the solver to generate a new input (i.e., x=0xdeadbeef). Concolic execution can keep exploring new paths by this way, thereby covering more code. However, due to the problem of path explosion and the complexity of path constraints, it is not scalable in real-world programs.



```
void func(int x) {
  if (x == 0xdeadbeef)
     bug()
  else
     bar()
}
```

Fig. 2. Example of concolic execution.

2.2 Motivation

Hybrid fuzzing, combining fuzzing and concolic execution, can complement the aforementioned limitations of each approach. In this section, we elaborate our research motivation by thoroughly analyzing the performance of three popular hybrid fuzzers, i.e., Driller, QSYM, and DigFuzz. We found that there are three problems to consider to implement an efficient and practical hybrid fuzzer.

(1) When should the concolic executor be launched?

Driller launches the concolic executor only when the fuzzer barely makes any progress for a certain period, i.e., when the fuzzer is stuck. More specifically, Driller runs AFL as the fuzzing component and invokes the concolic executor once the *pending_favs* attribute in AFL decreases to 0. However, this conservative strategy is problematic because the stuck state of a fuzzer is not a good indicator for launching concolic execution, as mentioned in [16]. The problem becomes more obvious for real-world programs, because the *pending_favs* attribute may not become 0 even after several hours' testing. In this case, the concolic executor has not been invoked throughout the fuzzing period, and thus the hybrid fuzzer degenerates into a pure fuzzer.

Different from Driller, QSYM and DigFuzz are unconcerned with the state of the fuzzer, that is, they do not check whether the current fuzzer is stuck or not. The concolic executor continuously synchronizes seed inputs from the fuzzer. Obviously, this strategy can benefit from concolic execution, thus we also adopt this strategy.

(2) What paths should be solved by concolic execution?

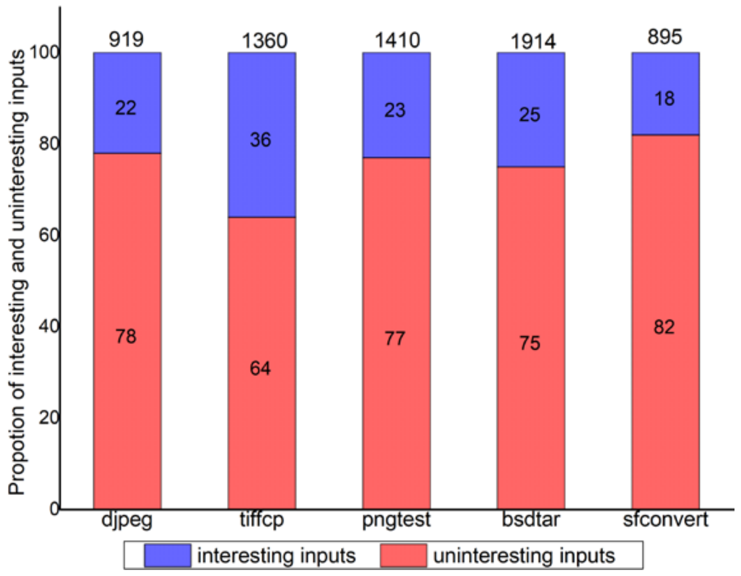
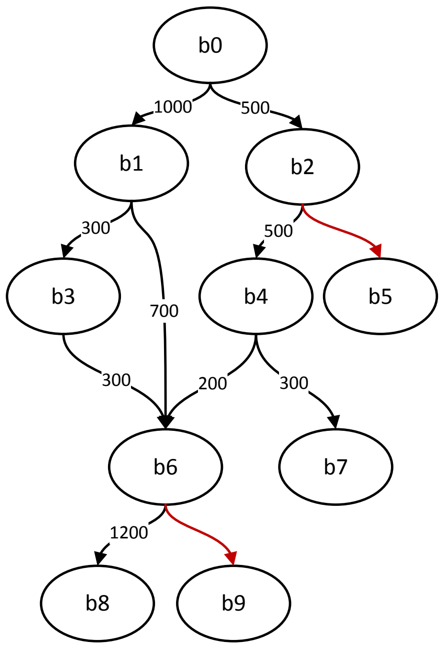


Fig. 3. Proportion of the interesting and uninteresting inputs for each binary. The numbers above the bars are the total number of inputs generated by QSYM.

When the concolic executor is launched, Driller and QSYM continuously feed all inputs generated by the fuzzer to the concolic executor, then the concolic executor flips all symbolic branches in the execution path triggered by each input in turn and generates new inputs. This radical strategy is also problematic, because many new inputs generated by concolic execution are uninteresting to the fuzzer and thus discarded. In fact, many branches with loose conditional constraints (e.g. ① and ② in **Fig. 1b**) can be quickly covered by a fuzzer, such as AFL. **Fig. 3** shows the proportion of interesting and uninteresting inputs generated by QSYM while testing 5 programs for 24 hours. On average only 24.8% of the inputs are considered interesting, that is, they are imported by AFL due to new branch coverage. The remaining 75.2% of the inputs are uninteresting and discarded, because they can be easily generated by lightweight fuzzing. In this case, concolic execution wastes considerable time for some useless works.

DigFuzz uses a relatively neutral strategy. It prioritizes paths that are most difficult for fuzzing to break through. Compared with Driller, DigFuzz is able to identify in time specific paths that block the fuzzer, thus achieving higher code coverage. However, according to our analysis, DigFuzz is only suitable for small programs (e.g., CGC binaries) that have fewer and shorter execution paths. Additionally, DigFuzz is difficult to apply to real-world programs, the reasons are as followings: 1) to quantify each path's difficulty, DigFuzz builds an execution tree, which will introduce a large runtime and memory overhead, especially for real programs. This limitation was also mentioned in [16]. 2) DigFuzz focuses on missed paths instead of branches, which may need a large amout of calulation. In fact, there are more missed paths than branches because the same uncovered branch may appear in multiple paths. As shown in **Fig. 4**, the uncovered branch b6 → b9 appears in three different paths (i.e., P1, P2, P3), and the uncovered branch b2 → b5 appears in one path (i.e., P4). To prioritize the hardest path (i.e., P3), DigFuzz has to traverse the execution tree and calculate the probability of these four paths. This is just a toy example to explain the problem. As the complexity of the program increases, there will be more and longer paths. The path quantification of DigFuzz will introduce considerable time overhead. Considering the above analysis, we borrow the idea of branch coverage of AFL to alleviate this problem. Only hardest-to-reach branch instead of path is prioritized, thus we do not have to build the execution tree. Section 3.2 will explain the design in detail. Note that this strategy may result in the loss of some paths with different contexts, but given the importance of time in fuzzing, we choose this less precise but lightweight approach.



Missed paths (DigFuzz):
P1: <b0, b1, b3, b6, b9>, P(P1)=0.0005
P2: <b0, b1, b6, b9>, P(P2)=0.00117
P3: <b0, b2, b4, b6, b9>, P(P3)=0.00033
P4: <b0, b2, b5>, P(P4)=0.002

Uncovered branches (EPfuzzer):
B1: b2 -> b5, hitcnt(N(B1))=500
B2: b6 -> b9, hitcnt(N(B2))=1200

Fig. 4. Program execution tree in DigFuzz. The number on the arrow indicates the hit count of the current branch, and the red arrow indicates the uncovered branch.

Our approach. Instead of flipping all conditional branches in the execution path blindly, we prioritize the uncovered branch that is really hardest to reach for the fuzzer based on the runtime coverage information and assign it to concolic execution.

(3) How to perform symbolic emulation and constraint solving efficiently?

It is well known that the main overhead of the concolic execution comes from the slow symbolic emulation and expensive constraint solving. QSYM's instruction-level symbolic emulation demonstrated its performance on real-world programs, but the concolic execution is still very slow during our testing. **Table 1** compares the time for symbolic emulation and constraint solving spent by QSYM, with the time for native execution and dynamic taint analysis (DTA) required for a single execution of five programs. We can see that the emulation time is still significantly slower. In particular, the emulation time for both *djpeg* and *tiffcp* exceeds 25 minutes (1500 s), thus many conditional branches may not be traversable in a limited time budget by concolic execution. The reason is that symbolic emulation needs to manage symbolic expressions whenever a tainted instruction is executed, resulting in a high overhead. Therefore, it is always desirable to further reduce the emulation time. In contrast, DTA takes less time, because it needs to tag only the memory addresses and registers affected by the input data propagated by running instructions instead of maintaining complex symbolic expressions.

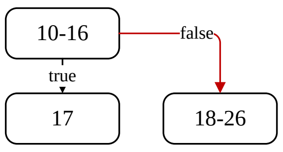
Table 1. Time for Native Execution, DTA, Qsym's Symbolic Emulation and Constraint Solving for a Single Execution.

Program	**Native(s)**	**DTA(s)**	**QSYM**	
			Emulation(s)	**Solving(s)**
djpeg+libjpeg	0.021	30.1	1568.3	726.7
tiffcp+libtiff	0.007	9.2	1643.8	175.2
pngtest+libpng	0.003	5.8	21.9	3.6
bsdtar+libarchive	0.006	13.9	59.1	12.4
sfconvert+audiofile	0.009	10.5	162.5	40.8

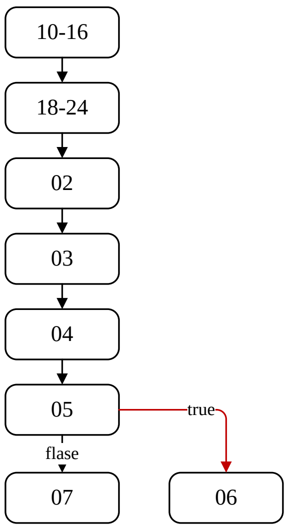
We noticed that when the current hybrid fuzzers invoke the concolic executor to generate new test cases, the concolic executor will mark all the input bytes as symbolic values and then build the symbolic expressions during the emulation execution. Actually, the number of symbolic instructions to be analyzed is positively related to the number of symbolic bytes. However, each conditional branch usually only depends on very few bytes and therefore we have no need to mark all the input bytes.

```
01 void bar( unsigned x, unsigned y, unsigned z) {
02    if(x < 2)
03       if(x + y < 3)
04          if(x+z != 0)
05             if(y > 1) /*complex condition check 2*/
06                y=1/(y-2);
07 }
08 void foo(char* buf) { /*many operations on buf here*/}
09 int main(int argc, char **argv) {
10    char buf[100];
11    FILE *fp = fopen(argv[1], "rb");
12    fread(buf, 1, 100, fp);
13    foo(buf[0:83])
14    unsigned int magic = 0;
15    memcpy(&magic, buf+84, 4);
16    if(magic != 0xdeadbeef)  /*complex condition check 1*/
17       return -1;
18    unsigned int x = 0;
19    memcpy(&x, buf + 88, 4);
20    unsigned int y = 0;
21    memcpy(&y, buf + 92, 4);
22    unsigned int z = 0;
23    memcpy(&z, buf + 96, 4);
24    bar(x, y, z);
25    return 0;
26 }
```

(a) A motivating example containing hard-to-read branches



(b) The path followed the true branch of check 1



(c) The path followed the false branch of check 2

Fig. 5. A motivating example. The numbers in boxes indicate the line numbers of the source code, and the red arrow indicates the uncovered branches.

To illustrate this problem more clearly, we take the code shown in the **Fig. 5a** as an example. The source code contains two conditional branches that are difficult for the fuzzer to break through (i.e., line 16 and 5). Assume that the initial input with 100 bytes triggers a path as shown in **Fig. 5b**. When the concolic executor is invoked, system calls related to IO operations (e.g., the *read* function) will be hooked, thereby marking the 100 bytes in the destination buffer *buf* as symbolic bytes. When the concolic execution reaches line 13, the *foo* function is called to perform some operations on the input byte offsets from 0 to 83. Since they are symbolic bytes, QSYM will spend considerable time emulating all instructions using symbolic bytes. When the execution reaches line 16, QSYM attempts to flip the branch by negating the current branch constraint and query the constraint solver to obtain an input (i.e., 0xdeadbeef) that can trigger the false branch. However, we can find that only four bytes (offsets 84-87) can affect the target branch at line 16, thus we do not need to symbolize byte offsets from 0 to 83. Similarly, for the path shown in **Fig. 5c**, to explore the true branch at line 5, we only need pay attention to the symbolic bytes corresponding to the variables x, y, and z. We will introduce the identification of relevant bytes in detail in section 3.3. It should be noted that Driller and DigFuzz, which use angr as the concolic executor, have the same problem with QSYM.

Our approach. Instead of marking the all input bytes as a symbolic values, we identify bytes relevant to the target branch to be flipped with DTA and symbolize these bytes to further reduce the overhead of symbolic emulation.

3. Design

In this section, we explain our decisions concerning the design of EPfuzzer.

3.1 Overview

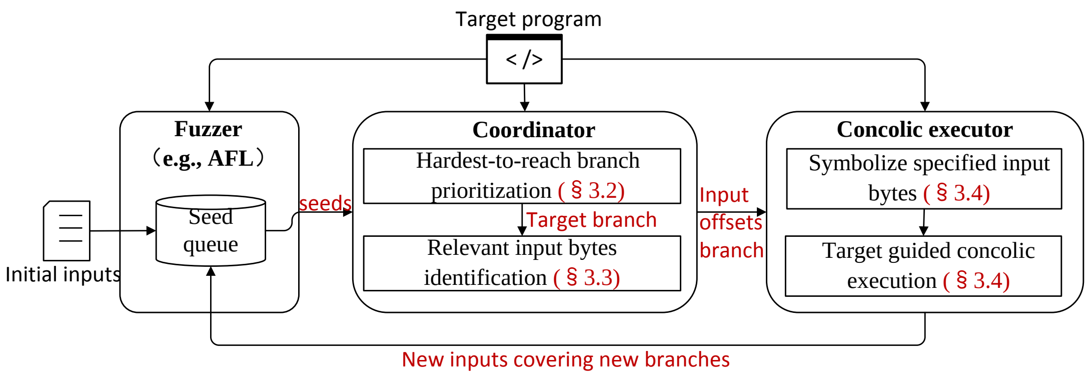

Target program
</>
Fuzzer
（e.g., AFL）
Seed queue
Initial inputs
seeds
Coordinator
Hardest-to-reach branch prioritization (§ 3.2)
Target branch
Relevant input bytes identification (§ 3.3)
Input offsets branch
Concolic executor
Symbolize specified input bytes (§ 3.4)
Target guided concolic execution (§ 3.4)
New inputs covering new branches

Fig. 6. Overview of EPfuzzer.

Fig. 6 shows the main components of the system and the overall workflow. Here, we summarize these components, the details will be covered in the following section.

Fuzzer: EPfuzzer can use any greybox fuzzing tools, such as AFL, honggfuzz [5], etc. It takes the initial inputs and the target program as inputs, generates new inputs through multiple mutation strategies (bit/byte flip, etc.) and stores them in the seed queue. Of course, our approach is orthogonal to other methods to improve coverage, thus fuzzers such as AFLFast and FairFuzz can also be feasible here.

Coordinator: As a bridge between the fuzzer and the concolic executor, coordinator is the core component of the system. It takes the seeds from the fuzzer as inputs, prioritizes the hardest-to-reach branch for the fuzzer by analyzing the coverage information (§ 3.2), and then uses DTA to identify the relevant bytes that affect the target branch to be flipped (§ 3.3). Finally, it outputs a triple, that is, the input that triggers the target branch, the byte offsets that affect the target branch, and the target branch to be flipped.

Concolic executor: It will symbolize the specified byte offsets according to the output of the coordinator, and then perform target guided concolic execution including symbolic emulation, target branch flipping, constraints collecting and solving, and finally terminate the execution (§3.4). Our concolic execution engine is implemented on the basis of QSYM, as it has been proven to work in real-world programs.

3.2 Hardest-to-reach Branch Prioritization

The purpose of this step is to prioritize the hardest-to-reach branch for the fuzzer and then assign it to the concolic execution. It can help to reduce the number of uninteresting inputs generated by Driller or QSYM. Additionally, it should be as lightweight as possible to avoid introducing much overhead like DigFuzz. Therefore, we prefer to choose the hardest-to-reach branch instead of the path.

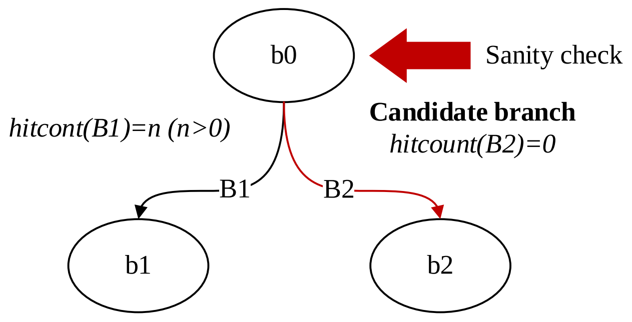


Fig. 7. Example of candidate branch.

The santity checks in the program are compiled into conditional jump instructions (e.g., jnz, jz, etc.) in the assembly code, and it is represented as a source basic block with two outgoing edges (true branch and false branch) in the control flow graph (CFG) as shown in Fig.7. To facilitate discussion, we introduce the following definitions.

Definition 1: Candidate branch. If a branch *B1* in the execution path has been explored many times, while its neighbor branch *B2* has never been explored, then the branch *B2* is considered a candidate branch that is hard for fuzzer to explore. The candidate branch can be formalized as all uncovered branches *Bi* satisfying the following conditions:

$$(hitcnt(Bi) = 0) \wedge (hitcnt(N(Bi)) = n(n > 0))$$

hitcnt(Bi) denotes the number of generated inputs which have exercised the branch *Bi*, and *N(Bi)* denotes the neighbor branch of *Bi*. In this case, we can obtain a set containing many candidate branches, denoted as *SC = {B1, B2, …, Bn}*. Next, we need to determine the priority of each branch in *SC* and assign the hardest-to-reach branch to concolic execution.

Definition 2: Hardest-to-reach branch. For a candidate branch *Bi* in *SC*, the intutition is that the greater the *hitcnt(N(Bi))* is, the harder it is for the fuzzer to explore branch *Bi*, so the branch *Bi* is a hardest-to-reach branch, if and only if:

$$\forall Bj \in SC \wedge Bj \neq Bi,\ \ hitcnt(N(Bi)) \geq hitcnt(N(Bj))$$

Definition 3: Target branch to be flipped. If a branch *Bi* is a hardest-to-reach branch, then its neighbor branch *N(Bi)* is the target branch to be flipped. In other words, we can use concolic execution to flip the *N(Bi)* to trigger the uncovered branch *Bi*.

As shown in **Fig. 4**, there are two uncovered branches *B1(b2 → b5)* and *B2(b6 → b9)*. Since *hitcnt(N(B1))* is less than *hitcnt(N(B2))*, thus the branch *B2(b6 → b9)* is deemed more difficult to explore and will be assigned to concolic execution first. We can find that our approach is more efficient and practical compared to DigFuzz. On one hand, we do not need to construct an execution tree, which is very expensive. On the other hand, we only calculate the hit count of only two branches (i.e., *B1* and *B2*), while DigFuzz has to calculate the probability of four paths (i.e., *P1, P2, P3* and *P4*).

Our implementation of hardest-to-reach branch prioritization is shown in Algorithm 1. This algorithm accepts two inputs and produces target branches to be flipped with the corresponding hit count. The two inputs are the target program to be analyzed and seed inputs retained by the fuzzer. First, we iterate all inputs from fuzzer to obtain hit count for each branch and store them in *HashMap* (lines 4-11). Then we analyze each conditional branch *Bi* in each trace *t* and obtain its neighbor branch *Bj* by CFG analysis (line 14). If the neighbor branch *Bj* is not covered, the current branch *Bi* will be added to the target branch set *ST* (line

12-21). Finally, we can prioritize the branch with the most hit count to flip based on *ST* information and generate an input to explore the uncovered branch.

Algorithm 1: Hardest-to-reach branch prioritization
Input: *program*: The binary to be analyzed
Input: *inputs*: The seed inputs retained by fuzzer
Output: *ST*: Target branch set with hit count corresponding to each branch

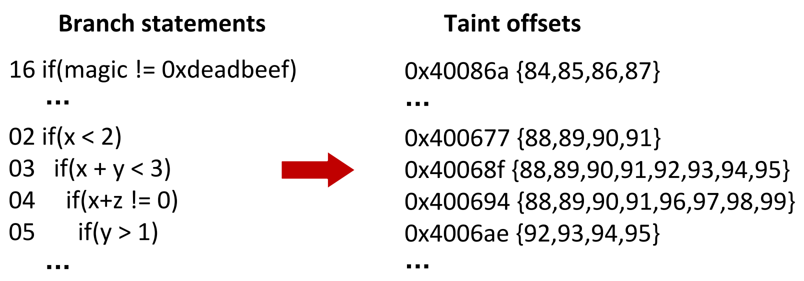
```
1  $HashMap \leftarrow \varnothing$
2  $Trace \leftarrow \varnothing$
3  $ST \leftarrow \varnothing$
4  for $i \in inputs$ do
5      $t \leftarrow GetTrace(program, i)$
6      $Trace \leftarrow Trace \cup t$
7      for $branch \in t$ do
8          $index \leftarrow Hash(branch)$
9          $HashMap[index]{++}$
10     end for
11 end for
12 for $t \in Trace$ do
13     for $Bi \in t$ do
14         $Bj \leftarrow GetNeighbor(Bi, CFG)$
15         $index_{Bi}, index_{Bj} \leftarrow Hash(Bi), Hash(Bj)$
16         if $HashMap[index_{Bj}] == 0$ then
17             $HitCount \leftarrow HashMap[index_{Bi}]$
18             $ST \leftarrow ST \cup (t, Bi, HitCount)$
19         end if
20     end for
21 end for
```

It should be noted that the branches in *SC* may contain some branches that we deem unlikely to help find bugs, thus we need to filter the undesired branches. For example, we are only interested in bugs in the target binary or library, so branches in other modules (e.g., libc) will not be tracked. QSYM tries to flip symbolic branches in all modules, thus possibly wasting much time. In addition, we also ignore some error-handling code. Take the code shown in **Fig. 8** as an example, the error-handling branch cannot be explored by the fuzzer quickly, but this branch is obviously undesired and exploring this branch does not contribute to bug finding. Therefore, we use a heuristic strategy similar to T-Fuzz [23] to tackle this. The error-handling branch is usually very short, because its logic returns or exits immediately after a detecting an error. We use the number of basic blocks following the conditional check as the length of the code paths and define a threshold value to tell an error-handling code path. This process is performed offline by static analysis, so it does not affect the overall performance of fuzzing.

```
if (tif->tif_diroff > (uint64)TIFF_INT64_MAX)
  {
       TIFFErrorExt(tif->tif_clientdata,module,"Can not read TIFF directory count");
       return(0);
  }
```

Fig. 8. Error-handling branch in libtiff.

3.3 Relevant Input Bytes Identification



```
16 if(magic != 0xdeadbeef)
   ...
02 if(x < 2)
03   if(x + y < 3)
04     if(x+z != 0)
05       if(y > 1)
   ...
```

```
0x40086a {84,85,86,87}
...
0x400677 {88,89,90,91}
0x40068f {88,89,90,91,92,93,94,95}
0x400694 {88,89,90,91,96,97,98,99}
0x4006ae {92,93,94,95}
...
```

Fig. 9. Taint offsets flow into branch statements.

As mentioned above, marking all the input bytes as symbolic values may introduce significant overhead. Since we already have the target branch from the last step, we need to symbolize only the bytes relevant to the target branch and treat the other bytes as concrete values thus effectively reduce the number of symbolic instructions. To this end, we use DTA to determine which input bytes affect the target branch. From **Table 1** we can see that the overhead of DTA is much lower than that of symbolic emulation. Therefore, we can improve overall performance especially in the following two cases: First, our approach works better for larger inputs. For example, AFL can accept an input less than 1 MB, but actually only several bytes may affect a certain branch. Second, our approach performs better for deeper branches, because it helps reduce many symbolic instructions before these branches.

We use the code in **Fig. 5a** to illustrate our method. First, we can obtain two hard-to-reach branches guarded by complex conditional check 1 and 2 by algorithm 1. Then, we need to identify bytes that affect these branches. **Fig. 9** shows the taint mapping between input bytes and the branch statements. The conditional check in line 16 is simple, and we just make variable *magic* (offsets 84~87) symbolic values. However, for the conditional check in line 5, we cannot make only variable *y* (offsets 92~95) as symbolic values, because it is a nested branch. If only variable *y* is mutated, the prior branch conditions (lines 2, 3, and 4) may be violated, thus resulting in the current branch being unreachable. We need to make all bytes that have data dependency on variable *y* as symbolic values, such as variable *x* and *z*. That is, to flip the branch of line 5, the 12 bytes (offsets 88~99) should be marked as symbolic values.

We present Algorithm 2 to achieve relevant input bytes identification, which relies on the following principle: if target branch *s* shares the same bytes directly or indirectly with its prior branch *r*, then *s* depends on bytes that affect *r*. This algorithm accepts three inputs and produces a set of byte offsets relevant to the target branch. The three inputs are the target program to be analyzed, the target branch to be flipped and the input that covers the target branch. First, we use DTA to obtain the taint mapping information *TaintMap* (line 3). Then, we get byte offsets *TO* directly used in target branch *Bt* (line 4). Finally, we query *TaintMap* backwards from *Bt* in a loop and add offsets that have data dependency on *TO* to the set *SO* (lines 7-16).

Algorithm 2: Relevant input bytes identification
Input: *program*: The binary to be analyzed
Input: *Bt*: The target branch aimed to flip
Input: *input*: The seed input that covers the target branch
Output: *SO*: input byte offsets which affect target branch

```
1 $SO \leftarrow \varnothing$
2 $Queue \leftarrow \varnothing$
3 $TaintMap \leftarrow DTA(input, program)$
4 $TO \leftarrow GetOffsets(TaintMap, Bt)$
5 $SO \leftarrow SO \cup TO$
6 add $TO$ to $Queue$
7 while $Queue$ not empty do
8   $TO \leftarrow Pop(Queue)$
9   for $branch \in TaintMap$ do  //Query TaintMap backwards from Bt
10    $CO \leftarrow GetOffsets(TaintMap, branch)$
11    if $TO \in CO$ then
12      $SO \leftarrow SO \cup CO$
13      Add $CO$ to $Queue$
14    end if
15  end for
16 end while
```

3.4 Target Guided Concolic Execution

After the steps in section 3.2 and 3.3, we can obtain a triple, that is, the target branch that needs to be flipped, the input that covers the target branch, and the relevant byte offsets that affect the target branch. When the concolic executor is launched, we make the specified bytes symbolic values and then perform symbolic emulation. When meeting a symbolic branch, we check whether it is the target branch to be flipped. If so, we flip the branch and generate an input that covers a new branch by solving collected path constraints. Thus, it can be guaranteed that we perform symbolic emulation only on less tainted instructions and solve the single target branch, thereby reducing the overhead of symbolic emulation. As the code shown in **Fig. 5**, assume magic = 0xdeadbeef, x = 1, y = 1, and z = 0, then the path in **Fig. 5b** will be covered. In this case, we mark the byte offsets from 88 to 99 as symbolic bytes and set the target branch 5→7. The concolic executor will flip the target branch, and replace the corresponding byte offsets of the original input with the solved value (x = 0, y = 2, z = 10) to generate a new input that can cover a new branch as shown in **Fig. 10**.

In addition, we observe that QSYM collects branch constraints in shared libraries such as libc, which can increase the complexity of path constraints and cause the constraint solver to fail. For example, QSYM cannot solve the path constraints of the conditional branch in line 5 of **Fig. 5a**. To this end, we ignore the branch constraints not in the target binary or library, because these constraints have no effect on path reachability. Additionally, to prevent the over-constraint problem as mentioned in [17], we keep optimistic solving which is a key feature of QSYM when the normal solving fails.

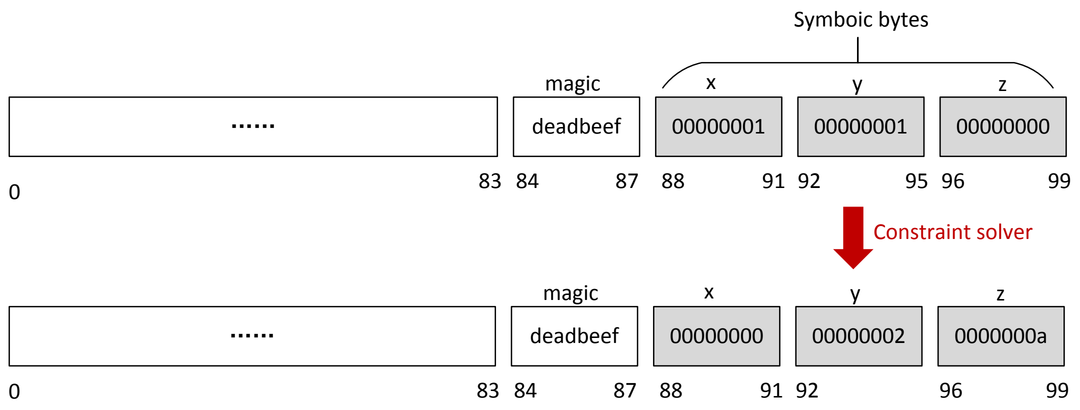


Fig. 10. Test case generation.

4. Evaluation

We implemented a prototype tool, EPfuzzer, in C++ and Python. The fuzzer module is based on AFL. Branch coverage information analysis and DTA are implemented based on Intel's instrument tool PIN [24], and CFG extraction is implemented based on IDAPython [25]. Target-guided concolic execution is based on the concolic execution engine QSYM.

4.1 Experiment Setup

To prove the effectiveness of our approach, we evaluate EPfuzzer and answer the following research questions:

RQ1: Can EPfuzzer achieve more code coverage?

RQ2: Can EPfuzzer find more bugs?

RQ3: How effective is EPfuzzer's new feature?

Baseline techniques to compare. We compare EPfuzzer (+1 master + 1 slave) with the original AFL (1 master + 1 slave) and QSYM (+1 master + 1 slave). Because Driller does not support real-world programs and DigFuzz is not publicly available now, we ignore these two tools. All the fuzzers run on a machine with Ubuntu 16.04 operating system, 128 G memory and 16 cores.

Target programs to test. We choose two existing benchmark datasets: LAVA-M and Google Fuzzer Test Suite (FTS). LAVA-M contains 4 GNU coreutils programs, each of which contains multiple known artificially injected bugs. FTS contains more complex programs than LAVA-M, and each program contains hard-to-reach lines of code or bugs. In addition, we also choose 5 real-world programs (xpdf, audiofile, nasm, libav, libtiff) which handle multiple file formats such as pictures, audios, videos, and asm files and have been tested in many previous works [10, 21, etc.]. For LAVA-M and FTS, we use the seeds they provide for initial inputs. For real-world programs, we choose 10 different seeds with an initial size of 1 KB.

4.2 Evaluation Results

(1) RQ1: Code Coverage Effectiveness

To answer RQ1, we evaluate the code coverage ability of fuzzers on FTS and real-word programs. Higher code coverage means a higher probability of finding bugs.

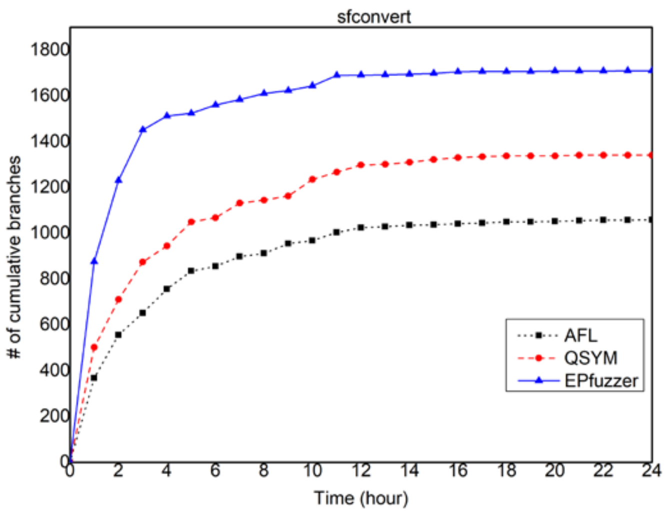
FTS. The purpose of this benchmark is to check whether a certain fuzzer can reach some specific code or bug locations. We can demonstrate the coverage ability of the fuzzer by observing the time it takes to reach these locations. The testing results is shown in **Table 2**. The second column displays the code location, and the third column shows the time required for each fuzzer to reach the location. The time-out is set to 24 hours. Except for the program *lcms*, EPfuzzer can reach the target location faster than QSYM and AFL. We analyze and find the main reason is the specified location of *lcms* is behind a branch that can be easily solved through fuzzer's random mutation, but EPfuzzer will prioritize the hard branch and ignore this one. However, for other code locations behind more complex branch conditions, EPfuzzer performs better than the other two fuzzers.

Table 2. Reaching time of EPfuzzer, QSYM and AFL to known bugs/code location

Program	**Location**	**Reaching time (hours)**		
		EPfuzzer	**QSYM**	**AFL**
guetzli	output_image.cc:398	2.55	3.16	3.82
lcms	cmsintrp.c:642	3.20	4.54	5.65
libarchive	archive_..._warc.c:537	6.15	6.82	8.14
Json	fuzzer-..._json.cpp:50	0.04	0.02	0.03
Libjpeg	jdmarker.c:659	8.75	15.18	T/O
Libpng	png.c:1035	5.46	5.81	7.28
vorbis	codebook.c:479	4.63	10.32	12.50

Table 3. Code coverage comparison with different fuzzers. x% in parentheses indicates the increased percentage compared to qsym

Program	**Project**	**LOC**	**Line coverage**			**Branch coverage**		
			AFL	**QSYM**	**EPfuzzer**	**AFL**	**QSYM**	**EPfuzzer**
pdftohtml	xpdf-4.02	60766	8179	8503	8922(+4.93%)	5398	5527	5762(+4.25%)
sfconvert	audiofile-0.3.6	9721	2323	2966	3295(+11.09%)	1059	1341	1710(+27.52%)
nasm	nasm-2.14.02	20357	904	1823	2836(+55.57%)	228	596	1332(+123.49%)
avconv	libav-12.3	205634	12385	14109	16754(+18.75%)	6218	6737	7396(+9.78%)
tiffcp	libtiff-4.0.10	30151	4951	5458	6315(+15.70%)	2874	3210	3905(+21.65%)
Average increase	-	-	-	-	21.21%	-	-	37.34%



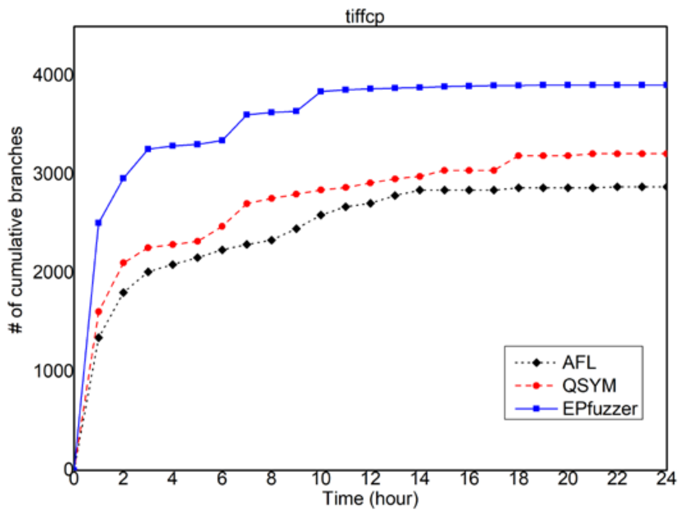


Fig. 11. Cumulative branch coverage on sfconvert and tiffcp.

Real-world programs. We test 5 real-world programs for 24 hours and use the afl-cov [26] tool to automatically record the results of the line and branch coverage. As shown in **Table 3**, EPfuzzer can cover more lines and branches than QSYM and AFL. In particular, compared to QSYM, the line coverage and branch coverage increased on average 21.21% and 37.34% respectively. It demonstrates that the hardest-to-reach branch prioritization strategy of EPfuzzer improves the efficiency of concolic execution. In addition, **Fig. 11** compares the branch coverage of *sfconvert* and *tiffcp* over time. It shows that EPfuzzer can cover branches faster than QSYM and AFL. It should be noted that we have similar results for the other three programs, but we do not list them all due to limited space.

(2) RQ2: Bug Finding Capability

To answer RQ2, we evaluate the bug-finding capability of fuzzers on LAVA-M and 5 real-word programs.

LAVA-M. We run each fuzzer for 5 hours on each program. Given the nature of the injected bugs, we equipped AFL with a dictionary (i.e., constants extracted from the binary). **Table 4** compares the the number of bugs found by each fuzzer. AFL performed the worst, which found a total of 8 bugs in all the programs. AFL(+Dict) performs better than AFL, because many bugs in LAVA-M are guarded by branches with magic bytes such as type ③ in **Fig. 1b**. However, AFL(+Dict) is not good at solving magic bytes that are not directly copied from the input but rather computed from the input such as type ④ in **Fig. 1b**, but it is easy for QSYM and EPfuzer to solve such constraints. EPfuzzer outperformed all the other tools and found 829 more bugs than QSYM in the program *who*, because it can quickly find branches that are hard for the fuzzer and solve these branches first. **Fig. 10** shows the cumulative number of bugs in *md5sum* and *who* found by QSYM and EPfuzzer over time. The bug discovery rate of QSYM becomes slower especially after 120 minutes, but EPfuzzer kept finding more bugs faster.

Table 4. LAVA-M Bugs Found by different fuzzers

Program	Listed bugs	EPfuzzer	QSYM	AFL	AFL (+Dict)
uniq	28	28(100%)	28(100%)	6(21%)	16(57%)
base64	44	44(100%)	44(100%)	1(2%)	15(34%)
md5sum	57	57(100%)	53(93%)	0(0%)	20(35%)
who	2136	2045(96%)	1216(57%)	0(0%)	58(3%)
Total	2265	2174	1345	8	109

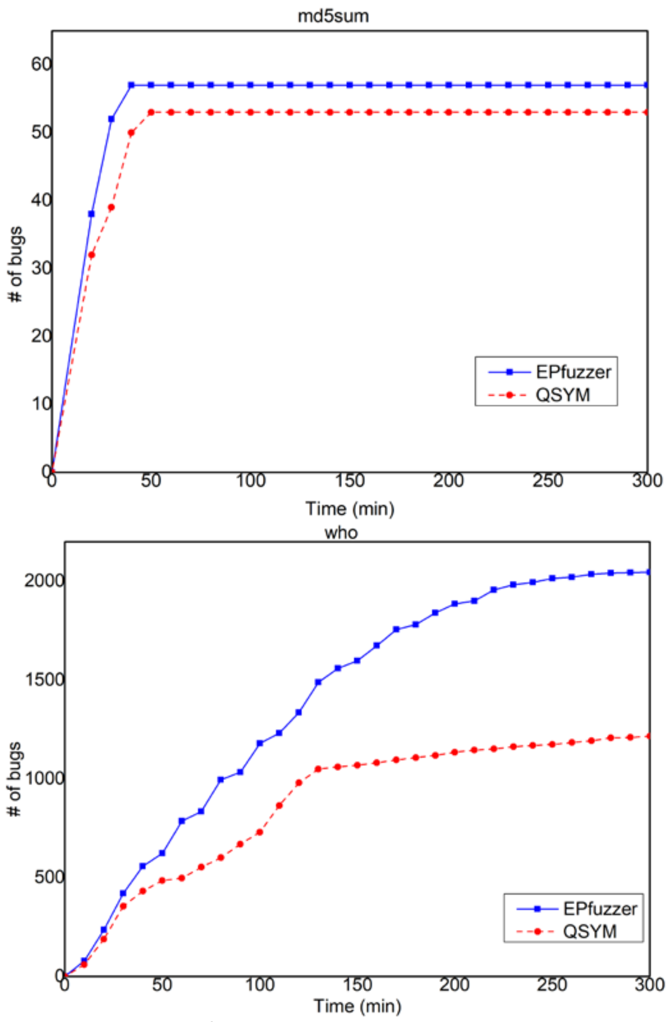


Fig. 12. The cumulative number of bugs detected in LAVA-M's md5sum and who over time.

Table 5. Number of bugs found by different fuzzers. Number in parentheses indicates number of crashes.

Project	**EPfuzzer**	**QSYM**	**AFL**
xpdf	3 (507)	1 (274)	1 (97)
audiofile	5 (113)	3 (84)	2 (33)
nasm	3 (352)	2 (120)	1 (68)
libav	5 (395)	3 (308)	2 (156)
libitff	1 (21)	0 (0)	0 (0)
Total	17 (1388)	9 (786)	7 (354)

Real-world programs. As shown in Table 5, EPfuzzer found 1388 crashes, and there were 17 unique bugs after deduplication with AddressSanitizer [27]. Seven bugs of them as shown in Table 6 have never been found before. The last column refers to the bug id assigned by the bug report platform (e.g., Bugzilla, Github) of the corresponding vendor. QSYM and AFL found 9 and 7 bugs respectively, and all the bugs found by these two fuzzers can also be found by EPfuzzer. The reason behind is that AFL has difficulty exploring branches guarded with many complex branch constraints so that many bugs cannot be reached. Although QSYM can also solve some complex constraints, it wasted much time in flipping branches blindly and performing expensive symbolic emulation, resulting in generating many uninteresting inputs.

Table 6. Unknown security bugs found in real-world programs

Project	Type	Bug ID
xpdf	Stack overflow	bug#41920
audiofile	Heap overflow	issue#56
nasm	use after free	bug#3392634
libav	Use after free	bug#1151
	Heap overflow	bug#1152
	Invalid memory access	bug#1153
	Use after free	bug#1154

(3) RQ3: New Feature of EPfuzzer

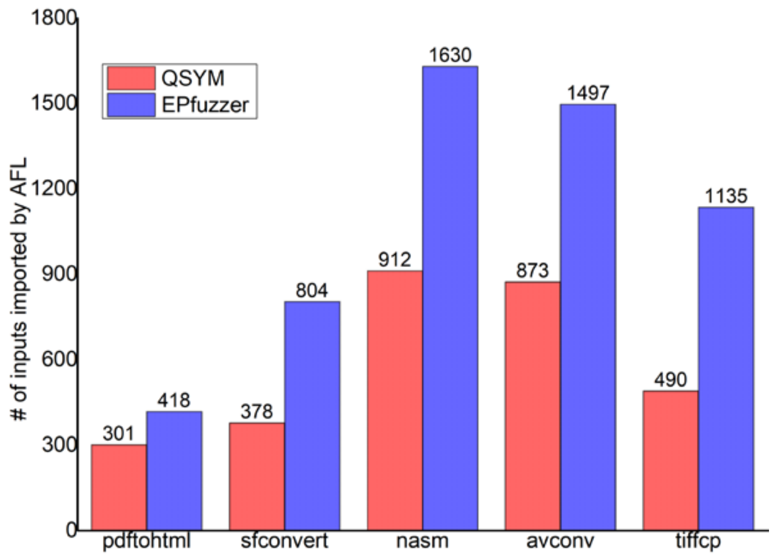


Fig. 13. The number of inputs imported by AFL.

To answer RQ3, we further evaluate the effectiveness of generated samples of EPfuzzer's concolic execution engine to better understand the improvements of EPfuzzer. We compare the number of interesting inputs (i.e., the inputs that contribute to branch coverage thus imported by AFL) generated by EPfuzzer's concolic execution engine with QSYM in the same time budget. As shown in **Fig. 13**, the percentage of interesting inputs generated by QSYM are approximately 44.7% less on average less than that generated by EPfuzzer. Three reasons are summarized as follows:

- EPfuzzer always prioritizes the hardest-to-reach branch for concolic execution, so inputs generated by EPfuzzer's concolic execution engine are most likely not generated by AFL's random mutation.
- EPfuzzer symbolizes only bytes affect the target branch, thus substantial emulation time is saved for the concolic execution engine to solve more complex branch constraints.
- EPfuzzer is unconcerned with conditional branches not in the target module such as libc, thus helping to save much solving time.

4.3 Discussion

This section discusses some limitations of our approach and some future improvements.

First, concolic execution cannot handle nonlinear constraints and floating-point operations. However, such constraints do not account for much of the program, so we do not consider them now.

Second, the current error-handling branch identification uses a heuristic strategy based on the number of basic blocks following the conditional check, which may incur a false positive. We can use the deep neural network to achieve this and improve the accuracy as introduced in [28].

Third, we use DTA based on PIN instrumentation to determine which input bytes affect a conditional branch. Although DTA is faster than symbolic emulation, it is still much slower than native execution. For this reason, we can use the fuzzing-driven taint inference method, as mentioned in [29], to further reduce the cost of DTA.

5. Related Work

A large body of related work has tried to improve the efficiency of fuzzing. In addition to aforementioned approaches besed on heuristic strategy or concolic execution, many researchers proposed tiant-guided fuzzing and learning-based fuzzing.

Taint-guided fuzzing. DTA can identify the dependencies between the program logic and input and thus it is usually used to help fuzzing to achieve more intelligent mutation. TaintScope [30] uses DTA to determine input bytes that flow into a security-sensitive function (e.g., malloc) and then mutate them. VUzzer [10] uses DTA to determine the bytes used for compare instructions and then replaces the bytes with the extracted constant values from binary to bypass some magic bytes checks. Similar to VUzzer, Angroa [31] also uses DTA to determine the bytes that affect the conditional branch and then uses gradient descent to solve the input that satisfies the condition constraints. In this paper, EPfuzzer leverages DTA to identify input bytes relevant to the target branch to be flipped thus reducing the number of symbolic instructions.

Learning-based fuzzing. Deep learning can learn a model automatically from many training samples and then be used to guide fuzzing to generate more effective samples. Learn&Fuzz [32] uses the sequence2sequence model to learn sample features from many pdf files. It can automatically generate high-quality pdf samples with high pass rate and coverage. Augmented-AFL [33] uses deep neural networks to learn which bytes in the input contribute to the coverage and achieve guided mutation. Neuzz [34] proposed a new program smoothing technique that uses neural networks to learn smooth estimates of complex branch behavior and then combines gradient-guided input generation methods to improve the effectiveness of fuzzing.

6. Conclusion

In this paper, we thoroughly investigate some state-of-the-art hybrid fuzzing systems and point out several fundamental limitations to them. We further propose a hardest-to-reach branch prioritization strategy. We adopt a more lightweight method to prioritize hardest-to-reach branches and identify input bytes relevant to target branch. We implement a prototype tool, EPfuzzer, based on the design and conduct comprehensive evaluation using three different datasets. The evaluation results show that compared with the state-of-the-art hybrid fuzzing system QSYM, the concolic execution in EPfuzzer contributes much more to the increased code coverage and increased number of discovered vulnerabilities.

Acknowledgments

This work was supported by National Key R&D Program of China under Grant 2019QY0501 and Soft Science Research Program of Henan Province under Grant 192102210128.

References

[1] The Heartbleed Bug. Accessed: Jan. 1, 2020. [Online]. Available: http://heartbleed.com/.

[2] WannaCry ransomware attack. Accessed: Jan. 1, 2020. [Online]. Available: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[3] Dirty COW Accessed: Jan. 1, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Dirty_COW.

[4] american fuzzy lop. Accessed: Jan. 1, 2020. [Online]. Available: http://lcamtuf.coredump.cx/afl/.

[5] Honggfuzz. Accessed: Jan. 1, 2020. [Online]. Available: https://github.com/google/honggfuzz.

[6] libFuzzer – a library for coverage-guided fuzz testing. Accessed: Jan. 1, 2020. [Online]. Available: https://llvm.org/docs/LibFuzzer.html.

[7] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of the 15th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, Feb.2008.

[8] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E:A platform for in-vivo multi-path analysis of software systems," in *Proc. of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Newport Beach, CA*, 265-278, Mar. 2011. Article (CrossRef Link)

[9] Shoshitaishvili Y, Wang R, Salls C, et al., "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Proc. of 2016 IEEE Symposium on Security and Privacy (SP). IEEE*, 138-157, 2016. Article (CrossRef Link)

[10] Rawat S, Jain V, Kumar A, et al., "VUzzer: Application-aware Evolutionary Fuzzing," in *Proc. of NDSS*, 17, 1-14, 2017. Article (CrossRef Link)

[11] Li Y, Chen B, Chandramohan M, et al., "Steelix: program-state based binary fuzzing," in *Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM*, 627-637, 2017. Article (CrossRef Link)

[12] Circumventing fuzzing roadblocks with compiler transformations. Accessed: Jan. 1, 2020. [Online].Available: https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.

[13] Aschermann C, Schumilo S, Blazytko T, et al., "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Proc. of NDSS*, 2019. Article (CrossRef Link)

[14] R. Majumdar and K. Sen, "Hybrid Concolic Testing," in *Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, MN*, May 2007. Article (CrossRef Link)

[15] Stephens N, Grosen J, Salls C, et al., "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proc. of NDSS*, 16(2016), 1-16, 2016. Article (CrossRef Link)

[16] Zhao L, Duan Y, Yin H, et al., "Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing," in *Proc. of NDSS*, 2019. Article (CrossRef Link)

[17] Yun I, Lee S, Xu M, et al., "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. of 27th {USENIX} Security Symposium ({USENIX} Security 18)*, 745-761, 2018.

[18] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *Proc. of IEEE Symposium on Security and Privacy (Oakland)*, 2016. Article (CrossRef Link)

[19] fuzzer-test-suite. Accessed: Jan. 1, 2020. [Online]. Available: https://github.com/google/fuzzer-test-suite.

[20] Böhme M, Pham V T, Roychoudhury A., "Coverage-based greybox fuzzing as markov chain,"

IEEE Transactions on Software Engineering, 45(5), 489-506, 2019. Article (CrossRef Link)

[21] Lemieux C, Sen K., "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," in *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 475-785, 2018. Article (CrossRef Link)

[22] Gan S, Zhang C, Qin X, et al., "Collafl: Path sensitive fuzzing," in *Proc. of 2018 IEEE Symposium on Security and Privacy (SP). IEEE*, 679-696, 2018. Article (CrossRef Link)

[23] Peng H, Shoshitaishvili Y, Payer M., "T-Fuzz: fuzzing by program transformation," in *Proc. of 2018 IEEE Symposium on Security and Privacy (SP). IEEE*, 697-710, 2018. Article (CrossRef Link)

[24] Pin - A Dynamic Binary Instrumentation Tool. Accessed: Jan. 1, 2020. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[25] IDAPython project for Hex-Ray's IDA Pro. Accessed: Jan. 1, 2020. [Online]. Available: https://github.com/idapython/src.

[26] afl-cov. Accessed: Jan. 1, 2020. [Online]. Available: https://github.com/mrash/afl-cov.

[27] ADDRESSSANITIZER. Accessed: Jan. 1, 2020. [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html.

[28] Song X, Wu Z, Cao Y, et al., "ER-Fuzz: Conditional Code Removed Fuzzing," *KSII Transactions on Internet & Information Systems*, 13(7), 2019. Article (CrossRef Link)

[29] Gan S, Zhang C, Chen P, et al., "GREYONE: Data Flow Sensitive Fuzzing,".

[30] Wang T, Wei T, Gu G, et al., "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. of 2010 IEEE Symposium on Security and Privacy. IEEE*, 497-512, 2010. Article (CrossRef Link)

[31] Chen P, Chen H., "Angora: Efficient fuzzing by principled search," in *Proc. of 2018 IEEE Symposium on Security and Privacy (SP). IEEE*, 711-725, 2018. Article (CrossRef Link)

[32] Godefroid P, Peleg H, Singh R., "Learn&fuzz: Machine learning for input fuzzing," in *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press*, 50-59, 2017. Article (CrossRef Link)

[33] Rajpal M, Blum W, Singh R., "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[34] She D, Pei K, Epstein D, et al., "Neuzz: Efficient fuzzing with neural program smoothing," in *Proc. of 2019 IEEE Symposium on Security and Privacy (SP). IEEE*, 803-817, 2019. Article (CrossRef Link)

YUNCHAO WANG received the M.S. degree in Computer Science and Technology from State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China, in 2016. He is currently pursuing the Ph.D. degree in Cyberspace Security with State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include reverse engineering and vulnerability discovery.

ZEHUI WU received the Ph.D. degree in Software Engineering from State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. He is currently a lecturer with State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include program analysis, reverse engineering and SDN security.

QIANG WEI received the Ph.D. degree in Computer Science and Technology from State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. He is currently a professor with State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include network security, industrial internet security and vulnerability discovery.

QINGXIAN WANG received the M.S. degree in the Department of Computer Science and Technology from Peking University. He is currently a professor with State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include network security, trusted computing and vulnerability discovery.