Rare Path Guided Fuzzing[*]

Seemanta Saha
University of California Santa Barbara
Santa Barbara, CA, USA
seemantasaha@ucsb.edu

Laboni Sarker
University of California Santa Barbara
Santa Barbara, CA, USA
labonisarker@ucsb.edu

Md Shafiuzzaman
University of California Santa Barbara
Santa Barbara, CA, USA
mdshafiuzzaman@ucsb.edu

Chaofan Shou
University of California Santa Barbara
Santa Barbara, CA, USA
shou@ucsb.edu

Albert Li
University of California Santa Barbara
Santa Barbara, CA, USA
albert_li@ucsb.edu

Ganesh Sankaran
University of California Santa Barbara
Santa Barbara, CA, USA
ganesh@ucsb.edu

Tevfik Bultan
University of California Santa Barbara
Santa Barbara, CA, USA
bultan@ucsb.edu

ABSTRACT

Starting with a random initial seed, fuzzers search for inputs that trigger bugs or vulnerabilities. However, fuzzers often fail to generate inputs for program paths guarded by restrictive branch conditions. In this paper, we show that by first identifying rare-paths in programs (i.e., program paths with path constraints that are unlikely to be satisfied by random input generation), and then, generating inputs/seeds that trigger rare-paths, one can improve the coverage of fuzzing tools. In particular, we present techniques 1) that identify rare paths using quantitative symbolic analysis, and 2) generate inputs that can explore these rare paths using path-guided concolic execution. We provide these inputs as initial seed sets to three state of the art fuzzers. Our experimental evaluation on a set of programs shows that the fuzzers achieve better coverage with the rare-path based seed set compared to a random initial seed.

CCS CONCEPTS

• **Software and its engineering** → **Software verification**; **Software reliability**.

KEYWORDS

Fuzz testing, Control flow analysis, Model counting, Probabilistic analysis, Concolic execution.

ACM Reference Format:
Seemanta Saha, Laboni Sarker, Md Shafiuzzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, and Tevfik Bultan. 2023. Rare Path Guided Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598136

1 INTRODUCTION

Fuzz testing has emerged as one of the effective testing techniques for achieving code coverage and finding bugs and vulnerabilities in software [8, 14, 25, 31, 33, 36, 38, 47]. Mutation-based fuzzers [25] and hybrid fuzzers [47] focus on improving coverage using mutation strategies or symbolic execution, respectively. Both of these techniques try to identify unexplored behaviors based on the inputs generated and branches covered during fuzzing. Note that, it may take a long time to generate a value that triggers a branch if the branch condition is very restrictive, and it is difficult to separate infeasible paths from feasible but rare paths via input mutation.

In this paper, we propose a lightweight whitebox analysis to identify rare paths in programs and then guide symbolic execution to generate inputs to explore these rare paths. Our approach avoids the shortcomings of mutation-based greybox fuzzers [8, 14, 25, 31, 33, 36] and hybrid fuzzers [38, 47] by generating inputs for rare paths beforehand, and it avoids the shortcomings of (symbolic execution based) whitebox fuzzers [17, 39] by reducing the cost of symbolic analysis.

We present a heuristic for identifying rare paths where we use control flow analysis, dependency analysis and model counting on branch constraints to transform a control flow graph to a probabilistic control flow graph. Then, we estimate path probabilities by traversing the probabilistic control flow graph and identify the rare (low-probability) paths.

Although our heuristic for estimating path probabilities and identifying rare paths is not sound and does not provide guarantees, our experimental results demonstrate that it is effective in generating seeds that trigger rare behaviors and it improves the coverage of existing fuzzers without even modifying them.

To improve the rare path analysis we introduce a new type of control flow paths (which we call II-paths) which is a combination

[*] This material is based on research sponsored by NSF under grants CCF-2008660, CCF-1901098 and CCF-1817242, and by DARPA under the agreement number N66001-22-2-4037. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598136

```
if ((int )id1 != (int )id2) {
    if ((int )id1 != (int )id3) {
      if ((int )id2 != (int )id3) {
        if ((int )id1 >= 0) {
          ...
              if ((int )r1 == 0) {
                ...
                    if ((int )max1 == (int )id1) {
                      if ((int )max2 == (int )id2) {
                        if ((int )max3 == (int )id3) {
                          if ((int )st1 == 0) {
                            if ((int )st2 == 0) {
                              ...
                                    if ((int )nl3 == 0) {
                                      if ((int )mode1 == 0) {
                                        if ((int )mode2 == 0) {
                                          if ((int )mode3 == 0) {
                                            tmp___5 = 1;
                                          } else {
                                            ......
```

Figure 1: A code fragment from pals_floodmax.3.1.ufo. BOUNDED-6.pals.c in the SV benchmarks.

of intra- and inter-procedural program paths, providing a balance between breadth first and depth first traversal of program paths. Our intuition is that II-paths-based analysis can generate more rare inputs compared to both intra- and inter-paths given a time budget.

We guide concolic execution using rare paths to generate inputs that trigger rare behaviors. As the last step of our approach, we provide the set of inputs generated by our analysis as the initial seed set to a fuzzer. This enables the fuzzer to explore the rare paths immediately, resulting in better coverage compared to randomly generated initial seed sets. Our approach can be integrated with all existing fuzzers that rely on initial seeds.

Our contributions in this paper are as follows:

- A novel technique for estimating path probabilities to identify rare paths in programs using a lightweight quantitative symbolic analysis.
- A new type of control flow paths (II-paths) to improve efficiency and effectiveness of the rare path analysis.
- Algorithms for path-guided concolic execution.
- Rare-path guided fuzzing approach where the initial seed set for a given fuzzer is generated with rare-path analysis.
- Experimental evaluation of the proposed techniques on existing fuzzers AFL++, FairFuzz and DigFuzz, demonstrating coverage improvement achieved by the proposed rare-path guided fuzzing approach, without even modifying the fuzzers.

The paper is organized as follows. We provide an overview in section 2. We explain program path analysis, heuristic to identify rare paths and input generation using the rare paths on section 3, 4 and 5, respectively. We discuss our implementation and experimental evaluations in section 6 and 7 respectively. We discuss related work in section 8 and provide our conclusions in section 9.

2 OVERVIEW

Fig. 1 is a code snippet from a program in the seq-mthreaded dataset of SV (software verification) benchmarks [6] used in software verification and testing competitions (SV-COMP and Test-Comp) [12, 13]. This code fragment contains a set of nested branch conditions leading to assignment of 1 to the *tmp__5* variable. If that variable is not set, the program will not exhibit a large set of behaviors. For a code fragment like this, it is difficult for fuzzers to generate a seed that triggers a program path reaching the assignment statement. Our experiments show that our approach improves performance of existing fuzzers (without modifying them) by generating seeds for rare paths that are unlikely to be covered by mutating randomly generated seeds.

Consider the running example in Fig. 2 which is a shortened version of a code structure found in libxml2. The main procedure of the program reads a string as an argument. It checks if the first 3 characters of the string is DOC or not. If the first 3 characters of the string is DOC, it parses the string starting from the 4th character. First, it goes inside the parse_cmt procedure and it checks if the 4th character is < or > and skips if it is. Then, the program comes back to the main procedure and goes inside the parse_att procedure. In the parse_att procedure, the program looks for the character sequence ATT. If it finds this sequence, it goes deeper into the program and executes more functionalities. To summarize, the program is trying to find two specific sequences of characters: first DOC and then ATT and if it can find these two sequences, it can execute more functionalities.

A mutation based fuzzer, such as AFL, starting with a random initial seed will require a lot of mutations to get to an input containing sequences DOC and ATT. We run AFL 5 times on the running example for an hour. We do not enable any additional options of AFL other than basic input output options. Length of the input is considered unknown, hence the input can be of any length. As an initial random seed/input we provide a single space. For 4 out of 5 times, AFL cannot generate an input containing sequences DOC and ATT. AFL can generate inputs such as DOC, DOC<, DOC>, DOCA etc. Though coverage guided mutation helps to reach these inputs, AFL is not able to generate the desired sequences as it mutates randomly and breaks already found sequences to inputs like DAC and DOCQ.

Now, let us explain how rare path analysis can guide a mutation-based fuzzer to achieve more coverage given a time budget. To perform rare path analysis on the running example program, we first extract the control flow graph and then we collect control flow paths of the program. At this point, we can use two well known existing techniques for control flow analysis to collect paths: intra-procedural control flow analysis and inter-procedural control flow analysis. Control flow graphs for the code in Fig. 2 are shown in Fig. 3.

First, we collect paths using intra-procedural control flow analysis (paths from 1 to 5 in Table 1). Among these paths, we find that path 4 is the rarest one. We identify rarity of the paths by computing path probability and we say that a path is the rarest if it has the lowest probability. Note that, to compute path probability, one can collect the path constraints using symbolic execution. In this paper, we do not use symbolic execution to collect path constraints. Instead we use a heuristic to compute path probabilities (discussed in section 4) that focuses on branch conditions and their selectivity.

After identifying the rare paths, we guide concolic execution (discussed in section 5) to generate inputs that trigger the rare paths. For example, for path 4 in Table 1, concolic execution generates the input DOC. We provide this input as the initial seed to AFL and we find that AFL can generate the sequences DOC and ATT within

```
char *CUR;
#define CMP3( s, c1, c2, c3 ) \
  ( ((unsigned char *) s)[ 0 ] == c1 && \
    ((unsigned char *) s)[ 1 ] == c2 && \
    ((unsigned char *) s)[ 2 ] == c3 )
int main(int argc, char **argv) {
  CUR = argv[1];
  if (CMP3(CUR, 'D', 'O', 'C')) {
    CUR = CUR + 3;
    parse_cmt();
    if(parse_att())
      /* go deeper */
  }
  return 0;
}
void parse_cmt() {
  if(*CUR == '<' || *CUR == '>')
    CUR++;
}
int parse_att() {
  if (CMP3(CUR, 'A', 'T', 'T'))
    return 1;
  return 0;
}
```

Figure 2: A code fragment based on the libxml file parser.c.

40 minutes (on average) whereas AFL with a random seed cannot generate these sequence in an hour.

We also collect paths using inter-procedural control flow analysis (paths from 20 to 43 in Table 1). Using our rare path analysis, we identify path 35 as the rarest one. Guiding concolic execution using path 35, the input generated is DOC<ATT. Providing this input as initial seed, fuzzer immediately explores the path covering sequences DOC and ATT.

Using inter-procedural control flow analysis, we can generate the rarest paths in the program. However, paths based on inter-procedural analysis also traverse parse_cmt which is not necessary to generate the desired sequences DOC and ATT that enable us to explore deeper behaviors. Although, for our small running example, analyzing the procedure parse_cmt will not waste too much analysis time, for larger real world cases like libxml2, focusing only on inter-procedural paths is likely be costly and can increase the cost of rare path analysis significantly.

To improve the effectiveness of rare path analysis (in order to generate a higher number of rare seeds within a given time budget) we introduce a new kind of control flow path in this paper which we call II-paths (discussed in section 3). II-paths subsume intra-procedural and inter-procedural control flow paths, and include more paths that combine their characteristics. All the paths in Table 1 are II-paths, where paths 1 to 5 are intra-procedural control flow paths, and paths 20 to 43 are inter-procedural control flow paths. Furthermore, paths 6 to 19 are also II-paths. Let us assume that, given a time budget, we can generate the paths from 1 to 20 only. Then, we will identify II-path 13 as the rarest one and concolic execution can generate the input DOCATT. As a result, we will able to generate an input containing sequences DOC and ATT while analyzing a relatively small number of paths.

3 PROGRAM PATHS

First step in rare-path guided fuzzing is identification of rare paths. The paths we identify are control flow paths that are generated by traversing control flow graphs of programs.

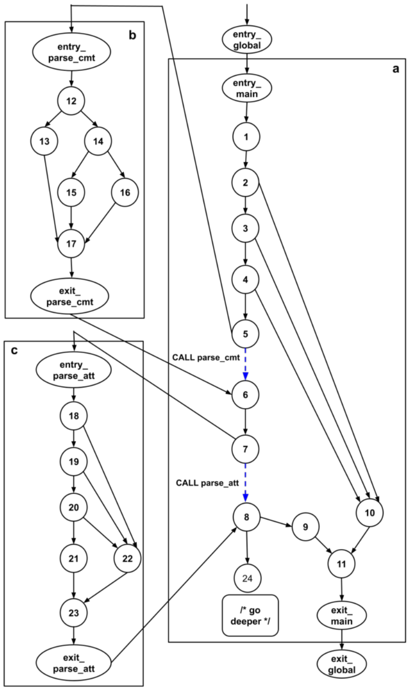


Figure 3: Control flow graphs for the code from Fig. 2.

3.1 Control Flow Graphs and Paths

Fig. 3 shows the control flow graphs for procedures shown in Fig. 2 in boxes a (main), b (parse_cmt) and c (parse_att). We define the control flow graph (CFG) [7] G_{pr} for a procedure *pr* as follows:

Definition 1. *A control flow graph for a procedure pr is a directed graph* $G_{pr} = (V, E)$ *where each vertex* $v \in V$ *represents a basic block of pr, and each directed edge* $e \in E : v \longrightarrow v'$ *represents a possible flow of control from vertex v to vertex* $v' \in E$ *. Control flow graph* G_{pr} *has a unique entry vertex entry-pr* $\in V$ *with no incoming edges and a unique exit vertex exit-pr* $\in V$ *with no outgoing edges. Furthermore, for each procedure call statement C to a procedure pr',* G_{pr} *contains a call vertex call-pr'*$_C \in V$ *and a return-site vertex return-pr'*$_C \in V$*, and an edge call-pr'*$_C \longrightarrow$ *return-pr'*$_C \in E$ *that represents the procedure call.*

An inter-procedural control flow graph represents control flow of the whole program by combining the control flow graphs of all procedures of the program.

Definition 2. *An Inter-Procedural Control Flow Graph (IP-CFG) for a program P,* $G_P^+ = (V, E)$*, contains the vertices and edges of the CFGs of all procedures in P, except the edges that correspond to procedure calls. Instead, for each procedure call statement C to a procedure pr in P,* G_P^+ *contains an edge from the call vertex to the entry vertex of the called procedure, call-pr*$_C \longrightarrow$ *entry-pr* $\in E$*, and an edge from the exit vertex of the called procedure to the return-site vertex for the call, exit-pr* $\longrightarrow$ *return-pr*$_C \in E$*, but it does not contain an edge between the call vertex and the return-site vertex, call-pr*$_C \longrightarrow$ *return-pr*$_C \notin E$*.* G_P^+ *also contains a vertex entry-global* $\in V$ *with no incoming edges (entry point of the program) and another vertex*

$exit\text{-}global \in V$ with no outgoing edges (exit point of the program), and connects them to the main procedure of the program P with edges $entry\text{-}global \rightarrow entry\text{-}main \in E$ and $exit\text{-}main \rightarrow exit\text{-}global \in E$.

Fig. 3 shows the IP-CFG for the code from Fig. 2 (the dashed edges are not part of the IP-CFG). For the call to procedure `parse_cmt`, there are two edges. One edge from call vertex 5 (which corresponds to *call-pr$_C$*) to *entry-parse_cmt* and one from *exit-parse_cmt* to return-site vertex 6 (which corresponds to *return-pr$_C$*).

We define intra- and inter-procedural control flow paths as follows:

Definition 3. *Given a control flow graph $G_{pr} = (V, E)$ for a procedure pr, an intra-procedural control flow path (intra-path) is a sequence of vertices $(v_1, v_2, v_3, \ldots, v_n)$ where $\forall i, v_i \in V, v_i \rightarrow v_{i+1} \in E$, $v_1 = entry\_pr$ and $v_n = exit\_pr$.*

Definition 4. *Given an inter-procedural control flow graph $G^+_P = (V, E)$ for a program P, an inter-procedural control flow path (inter-path) is a sequence of vertices $(v_1, v_2, v_3, \ldots, v_n)$ where $\forall i, v_i \in V, v_i \rightarrow v_{i+1} \in E$, $v_1 = entry\text{-}global$ and $v_n = exit\text{-}global$.*

Paths 1 to 5 in Table 1 correspond to all the intra-paths for the CFG of procedure *main*, and paths 20 to 43 in Table 1 are all the inter-paths for the IP-CFG of the whole program based on the control flow graphs shown in Fig. 3 for our running example. To save space, we only show the vertices with numeric labels in Table 1.

3.2 Intra-Inter Control Flow Paths (II-Paths)

We introduce a new type of control flow paths by combining both intra-paths and inter-paths. We call these paths intra-inter control flow paths (II-paths). Intuitively, for each procedure call, inter-paths have to choose a path inside the called procedure's CFG. On the other hand, intra-paths do not explore the CFGs of the called procedures. When visiting a procedure call statement, II-paths have the option to either behave like intra-paths (i.e., do not explore the CFG of the called procedure), or behave like inter-paths (i.e., explore the CFG of the called procedure).

In order to formally define II-paths we add back an extra edge to the IP-CFG between the call vertex *call-pr$_C$* and return-site vertex *return-pr$_C$* for each call statement C (as we had for the intra-procedural control flow graphs in Definition 1). We call the resulting control flow graph Extended Inter-Procedural Control Flow Graph (EIP-CFG):

Definition 5. *The Extended Inter-Procedural Control Flow Graph (EIP-CFG) for program P, denoted as $G^\star_P = (V', E')$, is defined using the IP-CFG $G^+_P = (V, E)$ of the program P, where $V' = V$ and $E \subseteq E'$. The only edges that are in E' and not in E are: For each procedure call statement C, a single edge between the call vertex call-pr$_C$ and the return-site vertex return-pr$_C$ is included in E', i.e., call-pr$_C$ $\rightarrow$ return-pr$_C$ $\in E'$ whereas call-pr$_C$ $\rightarrow$ return-pr$_C$ $\notin E$.*

Fig. 3 shows the EIP-CFG for our running example from Fig. 2 where the dashed edges are also part of the EIP-CFG. In the EIP-CFG, there are two edges from each call vertex *call-pr$_C$* for a procedure call: 1) to the entry vertex of called procedure *pr entry_pr*, i.e., edge *call-pr$_C$ → entry-pr* and 2) to the return-site vertex *return-pr$_C$*, i.e., edge *call-pr$_C$ → return-pr$_C$*. For example, in Fig. 3, the call vertex 5 has two outgoing edges corresponding to these two cases 1) 5 → *entry-parse_cmt* and 2) 5 → 6. As a result, whenever a call vertex is reached, there are two different paths to explore: 1) path taken via edge *call-pr$_C$ → entry-pr* which is similar to inter-paths, and 2) path taken via edge *call-pr$_C$ → return-pr$_C$* which is similar to intra-paths. Intuitively, every time a procedure call vertex is reached, II-paths can choose between considering or ignoring the control flow inside the called procedure. Whereas, intra-paths never explore the control flow of called procedures, and inter-paths always have to explore the control flow of the called procedures.

We define II-paths as follows:

Definition 6. *Given an EIP-CFG $G^\star_P = (V, E)$ for a program P, an intra-inter control flow path (II-path) is a sequence of vertices $(v_1, v_2, v_3, \ldots, v_n)$ where $\forall i, v_i \in V, v_i \rightarrow v_{i+1} \in E$, $v_1 = entry\text{-}global$ and $v_n = exit\text{-}global$.*

Again, let us consider the paths (listed in Table 1) of the EIP-CFG shown in Fig. 3 for our running example from Fig. 2. As we noted before, paths 1 to 5 in Table 1 are all the intra-paths for procedure *main*, and paths 20 to 43 in Table 1 are all the inter-paths for the program. Note that, based on the II-paths definition these paths are also II-paths. Furthermore, using the II-paths definition, in addition to II-paths from 1 to 5 and from 20 to 43, we now have additional II-paths from 6 to 19 where paths from 6 to 13 that ignore the control flow inside procedure `parse_cmt` but consider the control flow inside procedure `parse_att` and paths from 14 to 19 that ignore the control flow inside procedure `parse_att` but consider the control flow inside procedure `parse_cmt`.

Table 1: II-paths for the extended inter-procedural control flow graph shown in Fig. 3.

	Path	Probability
1	1 → 2 → 10 → 11	9.96×10^{-1}
2	1 → 2 → 3 → 10 → 11	3.98×10^{-3}
3	1 → 2 → 3 → 4 → 10 → 11	1.59×10^{-5}
4	1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 11	3.20×10^{-8}
5	1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 24	3.20×10^{-8}
6	1 → 2 → 3 → 4 → 5 → 6 → 7 → 18 → 22 → 23 → 8 → 9 → 11	3.19×10^{-8}
7	1 → 2 → 3 → 4 → 5 → 6 → 7 → 18 → 22 → 23 → 8 → 24	3.19×10^{-8}
8	1 → 2 → 3 → 4 → 5 → 6 → 7 → 18 → 19 → 22 → 23 → 8 → 9 → 11	1.27×10^{-10}
9	1 → 2 → 3 → 4 → 5 → 6 → 7 → 18 → 19 → 22 → 23 → 8 → 24	1.27×10^{-10}
10	1 → 2 → 3 → 4 → 5 → 6 → 7 → 18 → 19 → 20 → 22 → 23 → 8 → 9 → 11	5.10×10^{-13}
11	1 → 2 → 3 → 4 → 5 → 6 → 7 → 18 → 19 → 20 → 22 → 23 → 8 → 24	5.10×10^{-13}
12	1 → 2 → 3 → 4 → 5 → 6 → 7 → 18 → 19 → 20 → 21 → 23 → 8 → 9 → 11	2.05×10^{-15}
13	1 → 2 → 3 → 4 → 5 → 6 → 7 → 18 → 19 → 20 → 21 → 23 → 8 → 24	2.05×10^{-15}
14	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 8 → 9 → 11	1.28×10^{-10}
15	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 8 → 24	1.28×10^{-10}
16	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 8 → 9 → 11	1.27×10^{-10}
17	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 8 → 24	1.27×10^{-10}
18	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 8 → 9 → 11	3.17×10^{-8}
19	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 8 → 24	3.17×10^{-8}
20	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 18 → 22 → 23 → 8 → 9 → 11	1.27×10^{-10}
21	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 18 → 22 → 23 → 8 → 24	1.27×10^{-10}
22	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 18 → 19 → 22 → 23 → 8 → 9 → 11	5.10×10^{-13}
23	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 18 → 19 → 22 → 23 → 8 → 24	5.10×10^{-13}
24	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 18 → 19 → 20 → 22 → 23 → 8 → 9 → 11	2.04×10^{-15}
25	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 18 → 19 → 20 → 22 → 23 → 8 → 24	2.04×10^{-15}
26	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 18 → 19 → 20 → 21 → 23 → 8 → 9 → 11	8.19×10^{-18}
27	1 → 2 → 3 → 4 → 5 → 12 → 13 → 17 → 6 → 7 → 18 → 19 → 20 → 21 → 23 → 8 → 24	8.19×10^{-18}
28	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 18 → 22 → 23 → 8 → 9 → 11	1.27×10^{-10}
29	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 18 → 22 → 23 → 8 → 24	1.27×10^{-10}
30	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 18 → 19 → 22 → 23 → 8 → 9 → 11	5.08×10^{-13}
31	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 18 → 19 → 22 → 23 → 8 → 24	5.08×10^{-13}
32	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 18 → 19 → 20 → 22 → 23 → 8 → 9 → 11	2.03×10^{-15}
33	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 18 → 19 → 20 → 22 → 23 → 8 → 24	2.03×10^{-15}
34	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 18 → 19 → 20 → 21 → 23 → 8 → 9 → 11	8.16×10^{-18}
35	1 → 2 → 3 → 4 → 5 → 12 → 14 → 15 → 17 → 6 → 7 → 18 → 19 → 20 → 21 → 23 → 8 → 24	8.16×10^{-18}
36	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 18 → 22 → 23 → 8 → 9 → 11	3.16×10^{-8}
37	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 18 → 22 → 23 → 8 → 24	3.16×10^{-8}
38	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 18 → 19 → 22 → 23 → 8 → 9 → 11	1.26×10^{-10}
39	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 18 → 19 → 22 → 23 → 8 → 24	1.26×10^{-10}
40	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 18 → 19 → 20 → 22 → 23 → 8 → 9 → 11	5.06×10^{-13}
41	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 18 → 19 → 20 → 22 → 23 → 8 → 24	5.06×10^{-13}
42	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 18 → 19 → 20 → 21 → 23 → 8 → 9 → 11	2.03×10^{-15}
43	1 → 2 → 3 → 4 → 5 → 12 → 14 → 16 → 17 → 6 → 7 → 18 → 19 → 20 → 21 → 23 → 8 → 24	2.03×10^{-15}

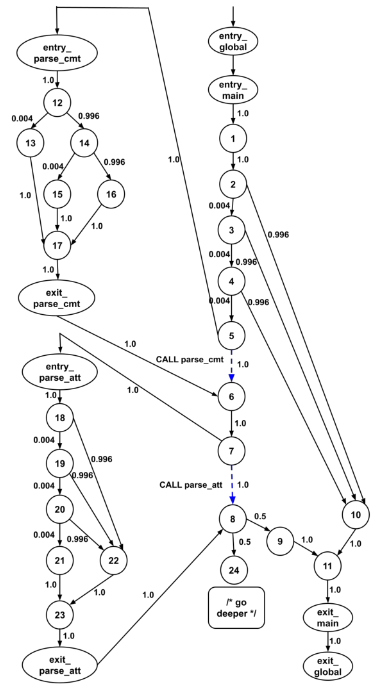


Figure 4: Probabilistic inter-procedural control flow graph for the code from Fig. 2.

4 IDENTIFYING RARE PATHS

In this section, we describe construction of a probabilistic control flow graph to compute path probabilities. Then, we identify the rare paths based on path probabilities.

4.1 Path Probability

Given a program P, let i denote the input for the program, and I denote the domain of inputs (i.e., $i \in I$). Given a path t in program P, the goal of path probability analysis is to determine how likely it is to execute the path t. We do this by determining the likelihood of picking inputs that result in the execution of path t. In order to determine the likelihood of picking such inputs, we compute the probability of picking such inputs if the inputs are chosen randomly. We define $\mathcal{P}(P, t)$ as:

Definition 7. *$\mathcal{P}(P, t)$ denotes the probability of executing the path t of program P where the input i of the program P is randomly selected from the input domain I.*

To compute path probability, we assume that inputs are uniformly distributed. However, one can extend our technique for path probability computation by integrating usage profile [18], used in other probabilistic analysis techniques.

Path probabilities can be computed using quantitative extensions of symbolic execution such as probabilistic and statistical symbolic execution [18, 21]. However, these symbolic execution based techniques have a high computation complexity and poor scalibility due to the cost of path constraint solving and model counting over an exponentially increasing number of paths. Recently, a new heuristic-based technique has been proposed for probabilistic reachability analysis [37], which reduces the complexity of probabilistic analysis using a concept called branch selectivity. In this paper, we focus on computing path probabilities using branch selectivity instead of computing reachability probabilities of program statements using a Discrete-Time Markov Chain model as in [37].

4.2 Probabilistic Control Flow Graph

To compute path probabilities, we introduce the concept of the probabilistic control flow graph (Prob-CFG). One can interpret a probabilistic control flow graph as a weighted control flow graph where the edge weights are probability estimates. Prob-CFG is not a discrete time Markov chain; the sum of probabilities of the outgoing edges of a Prob-CFG node does not always add up to one.

Prob-CFG $PG_P^\star$ for a program P is constructed using the extended inter-procedural control flow graph (EIP-CFG) $G_P^\star$ for P. We define the probabilistic control flow graph $PG_P^\star$ as follows:

Definition 8. *Given a program P and its EIP-CFG $G_P^\star = (V, E)$, the probabilistic control flow graph $PG_P^\star$ for program P is defined as $PG_P^\star = (V, E, F)$ where the set of vertices and edges for $PG_P^\star$ are same as the set of vertices and edges of $G_P^\star$, and F is a function $F : E \longrightarrow [0, 1]$ that assigns a probability score to each edge in E.*

As we describe below, we use dependency analysis and branch selectivity to compute probability scores of the edges in probabilistic control flow graphs.

Dependency Analysis. A branch condition in the program is input dependent if the evaluation of the branch condition depends on the value of the program input. Given a program and input(s) to the program, we use static dependency analysis to identify the input dependent branch vertices in the control flow graph. Static dependency analysis over-approximates the set of input-dependent branch vertices. As a result, the path probability we compute is an estimation of the actual path probability. Anyway, we use branch selectivity, a heuristic to estimate path probability.

Branch Selectivity. To compute the probability for each edge in the control flow graph, we use branch selectivity. We use the definition of branch selectivity $\mathcal{S}(b)$ as in [37]:

Definition 9. *Given a branch condition b, let D_b denote the Cartesian product of the domains of the variables that appear in b, and let $T_b \subseteq D_b$ denote the set of values for which branch condition b evaluates to true. Let $|D_b|$ and $|T_b|$ denote the number of elements in these sets, respectively. Then, $\mathcal{S}(b) = \frac{|T_b|}{|D_b|}$ and $0 \leq \mathcal{S}(b) \leq 1$.*

We compute $|T_b|$ using a model counting constraint solver. Branch selectivity gets closer to 0 as the number of values that satisfy the branch condition decreases and gets closer to 1 as the number of values that satisfy the branch condition increases.

We define the probability score function F for the probabilistic control flow graph $PG_P = (V, E, F)$ using the combination of dependency analysis and branch selectivity as follows:

- If there is only one edge starting from a vertex v to u, then the probability of the edge $e : v \longrightarrow u$ is 1, i.e, $F(e) = 1$.

- If v is a vertex with branch condition b, there are two edges from source vertex v: $e_1 : v \longrightarrow u_1$ and $e_2 : v \longrightarrow u_2$, where e_1 is the true evaluation and e_2 is the false evaluation of branch condition b:
  – If branch condition b is dependent on program input, then probability of edge e_1 is the branch selectivity, $\mathcal{S}(b) = \frac{|T_b|}{|D_b|}$ and the probability of edge e_2 is $1-\mathcal{S}(b)$, i.e., $F(e_1) = \mathcal{S}(b)$ and $F(e_2) = 1 - \mathcal{S}(b)$.
  – If branch condition b is not dependent on program input, then probability of both edges e_1 and e_2 is 1, i.e., $F(e_1) = F(e_2) = 1$.
- Probabilities of edges that have a call vertex as their source e_1 : *call-pr*$_C \longrightarrow$ *entry-pr* and e_2 : *call-pr*$_C \longrightarrow$ *return-pr*$_C$ are 1, i.e., $F(e_1) = F(e_2) = 1$.

By adding probabilities to all the edges in a control flow graph, we transform it to a probabilistic control flow graph. For the cases where we assign probability score function F as $F(e_1) = F(e_2) = 1$, only one of the two edges is viable during the program execution, and such edges do not influence path probability.

Consider the EIP-CFG in Fig. 3. Each branch vertex is associated with a branch condition. For example, vertex 2 is associated with branch condition `CUR[0] = D`. We consider that the inputs are uniformly distributed and domain for each character in a string has 256 values. Branch selectivity $\mathcal{S}$ for the branch condition at vertex 2 is $\frac{1}{256} \equiv 0.004$. Hence, probability for the edges $2 \longrightarrow 3$ is 0.004 and probability for the edge $2 \longrightarrow 10$ is $1 - 0.004 = 0.996$. We add all the edge probabilities to the EIP-CFG $G_P^\star$ in Fig. 3 and construct the probabilistic EIP-CFG $PG_P^\star$, shown in Fig. 4. Once we construct the probabilistic control flow graph $PG_P^\star$, we can compute path probabilities as follows:

Definition 10. *Given a control flow path t of length n for program P which corresponds to a sequence of vertices $\{v_1, v_2, v_3, \ldots, v_n\}$ in the probabilistic control flow graph $PG_P^\star = (V, E, F)$, then path probability $\mathcal{P}(P,t)$ for path t is computed as*

$$\mathcal{P}(P,t) = \prod_{i=1}^{n-1} F(v_i, v_{i+1})$$

Path probabilities computed for the II-paths for our running example using the probabilistic control flow graph in Fig. 4 are shown in Table 1.

4.3 Rare Paths

We call a program path a rare path if it is unlikely to be executed when the program input is randomly chosen. Since there may be an unbounded number of paths in a program, given a depth bound b, we identify the set of k rare paths among all paths with length less than or equal to b.

In order to identify a set of rare paths R with size k for a given execution depth b, we compute path probabilities of all paths of length less than or equal to b and choose the k paths with the smallest path probabilities.

For example, traversing through the probabilistic control flow graph in Fig. 3 we generate 43 II-paths and compute corresponding path probabilities as shown in Table 1. Now, if we sort these paths in an ascending order based on the path probability and pick the set of rare paths R for $k = 3$, we identify paths 34, 35 and 26 as the paths in the rare path set R. A fuzzer that randomly generates inputs would be very unlikely to explore these rare paths.

5 INPUT GENERATION FOR RARE PATHS

The analysis we described above results in the set R of k rare paths in the program. However, it does not identify k inputs that can trigger these rare paths in the program. The input generation process we describe in this section identifies inputs to trigger the rare paths in the set R. In order to generate the set of rare inputs I_R for the set of rare paths R we guide concolic execution using each rare path $t_R \in R$ and generate input i_R for each t_R (if path t_R is a feasible execution path). We add all these inputs to the set of rare inputs I_R.

Note that, the rare paths we compute are based on an estimation of path probability and some of the rare paths might not be feasible. But, concolic execution captures the original program execution semantics. Hence, if a rare path is not feasible, it will be eliminated in the input generation step using concolic execution.

We use path-guided concolic execution to collect path constraints for a rare path. We then use a SMT solver to solve the path constraints and generate the input that can be fed to the program to execute the rare path. We provide two different algorithms for path-guided concolic execution for input generation: 1) Inter-path guided concolic execution, 2) II-path guided concolic execution.

Algorithm 1 IP-GCE(P, t_R)
Takes a program P and an inter-procedural path t_R as input and generates an input for P to execute the path t_R

```
input ← Random()
t_C ← Execute(P, input)
index ← 2
while index < Len(t_C) ∧ index < Len(t_R) do
    if t_C(index) ≠ t_R(index) then
        path_cond ← NegatedPath(t_C, index)
        if IsFeasible(path_cond) then
            input ← Solve(path_cond)
            t_C ← Execute(P, input)
        else
            return input
    index ← index + 1
return input
```

5.1 Inter-path Guided Concolic Execution

For inter-path guided concolic execution (IP-GCE), we run the program on a concrete random input and generate the corresponding inter-path t_C. In order to generate input for the rare path t_R, we compare all branches for t_C and t_R in the same order. If there is a mismatch between any of the branches, we negate the branch and solve it to check feasibility of the path negating the branch. If the path is feasible, we solve the path constraint and generate the new input. We then execute the program using the new input and update t_C by the inter-path generated by the new input. The process continues as long as there are branches left to compare both in t_C and t_R or there are no branches that can lead to a feasible path. At the end of the process, the input is the input that will either take path t_R or take a path that is close to the rare path t_R if t_R is not feasible.

Algorithm 2 IIP-GCE(P, t_R)
Takes a program P and an II-path t_R as input and generates an input for P to execute a path that has high overlap with t_R

```
1:  $input \leftarrow$ RANDOM()
2:  $t_C \leftarrow$ EXECUTE($P$, $input$)
3:  $max\_overlap \leftarrow$ OVERLAP($t_C$, $t_R$)
4:  $max\_input \leftarrow input$
5:  $index \leftarrow 1$
6:  while $index <$ LEN($t_C$) do
7:      if ISBRANCH($t_C(index)$) $\wedge$ DIFFER($t_C(index)$, $t_R$) then
8:          $path\_cond \leftarrow$ NEGATEDPATH($t_C$, $index$)
9:          if ISFEASIBLE($path\_cond$) then
10:             $input \leftarrow$ SOLVE($path\_cond$)
11:             $t_C \leftarrow$ EXECUTE($P$, $input$)
12:             $overlap \leftarrow$ OVERLAP($t_C$, $t_R$)
13:             if $overlap > max\_overlap$ then
14:                 $max\_overlap \leftarrow overlap$
15:                 $max\_input \leftarrow input$
16:             else
17:                 $input \leftarrow max\_input$
18:                 $t_C \leftarrow$ EXECUTE($P$, $input$)
19:     $index \leftarrow index + 1$
20: return $input$
```

Algorithm 1 shows the process of guiding concolic execution using rare inter-path. EXECUTE executes the program P first on a random input and returns the corresponding execution path t_C. The algorithm looks for the first vertex where t_C and t_R differ (all paths start with the same vertex). NEGATEDPATH(t_C, $index$) generates a path constraint corresponding to the path t_C where the branch condition between the vertex $index - 1$ and $index$ is negated and all the branches before $index - 1$ remain the same. ISFEASIBLE checks the feasibility of a given path constraint and SOLVE generates an input value satisfying the given path constraint.

5.2 II-Path Guided Concolic Execution

In this section we discuss II-Path guided concolic execution (IIP-GCE) which can also handle intra-paths since intra-paths are also II-paths. An II-path can be infeasible, so it may not represent a concrete execution path. Hence, IIP-GCE algorithm is not guaranteed to generate an input exercising the given II-path. IP-GCE algorithm we discussed in the previous section uses branch matching and branch negation for mismatched branches, but this approach is not sufficient for guiding the concolic execution to explore the rare II-paths since II-paths may not represent a concrete execution path.

Similar to the *IP-GCE* algorithm, in the *IIP-GCE* algorithm (Algorithm 2), we first run the program on a concrete random input and collect the execution path t_C. Note that, there may be branches in t_C that are in a procedure that is not explored in the input II-path t_R. In such situations, we compare the inputs that trigger both the branch and its negation, and see which one creates an execution path that overlaps more with t_R (i.e., increases the number of vertices that are common in both), and then we pick the branch which results in higher overlap with t_R.

Lines 1-5 in Algorithm 2 generate the initial concrete path t_C with a random input, and calculate the initial overlap between t_C and t_R using the function OVERLAP. The while loop in lines 6-19 iterates over the nodes in t_C. It looks for branch nodes in t_C that differ from the corresponding branch node in t_R. The function DIFFER returns true under two conditions: 1) there is a branch in t_R that corresponds to complement of $t_C(index)$ (i.e., t_R and t_C take different branches for the same branch statement), or 2) there is no branch in t_R that corresponds to the branch $t_C(index)$ (this branch node in t_C corresponds to a branch in a procedure that was not explored in t_R). In both of these cases we negate the branch condition at $t_C(index)$ and see if we can improve the overlap between t_C and t_R, and update the input and t_C if the overlap can be improved. Note that, if the overlap cannot be improved, then the input is restored to the previous input in lines 17-18.

Algorithm 2 makes a single pass on the branches in t_C without backtracking and therefore it is not guaranteed to find an execution that maximizes the overlap between final t_C and t_R. Looking for maximum overlap would require a search on all execution paths, resulting in path explosion that we have to avoid for scalability.

For the running example, guiding concolic execution using path 35, input generated is `DOC<ATT`. whereas guiding concolic execution using path 34, we find out that path 34 is infeasible. Hence, path-guided concolic execution algorithms we provide does not only generate inputs but also checks feasibility of the rare paths. Even though our techniques for identifying rare paths in the program is a heuristic approach, infeasible rare paths will be always filtered out in the input generation phase. The inputs we generate are always valid inputs and they help fuzzer in exploring rare program paths.

6 IMPLEMENTATION

We implement our techniques (rare path analysis and path-guided concolic execution) to analyze programs written in the C programming language. We extract branch conditions and control flow graph for a program using the concolic execution tool CREST [16] and underlying program transformation tool CIL [34]. To collect branch conditions from the program, we modify the OCaml code in CIL. We transform the branch conditions into constraints in SMT-LIB format. To model count the branch constraints, we use Automata-based Model Counter (ABC) [9]. To identify branches that are input-dependent, we perform dependency analysis using CodeQL [3], a code analysis engine.

After extracting the control flow graph and collecting the model counts for the input-dependent branches, we transform the control flow graph to a probabilistic control flow graph. We write python scripts to traverse the probabilistic control flow graph and collect intra-, inter-, and II-paths.

We guide concolic execution tool CREST [16] using the rare paths we collect from our control flow analysis. We implement algorithms IP-GCE and IIIP-GCE in C on top of the existing concolic search strategies in CREST. We use existing coverage-guided fuzzers AFL++ [19] and FairFuzz [25] without modification. We implement the proposed technique of the fuzzing tool DigFuzz [47] using AFL++ and QSym [46]. To collect edge coverage we use `afl-showmap` as used in [41].

Note that, as with any approach that builds on other techniques, our approach does have limitations that are due to its building blocks (for example, the program transformation tool CIL cannot support some program constructs and model counting techniques have limitations similar to constraint solvers used in symbolic execution). These limitations can be lifted with progress in the building blocks we use.

7 EXPERIMENTAL EVALUATION

To evaluate our techniques for rare path-guided fuzzing we experiment on three different sets of programs. The first set of programs we use are from the seq-mthreaded directory [6] of the SV benchmark used in SV-COMP and Test-Comp [12, 13]. We run our experiments on 62 programs from the seq-mthreaded directory since, in prior work [26], they are identified as a set of programs for which randomly-picked seed inputs are ineffective. These programs contain lots of nested restricted branch conditions which are challenging to handle for existing fuzzers.

The second set of programs has been used in the experimental evaluation of a parser-directed fuzzer [32]. These programs deal with structured inputs and contain a lot of restrictive branch conditions. We add *calculator* [2] in this set which contains numerous restrictive branch conditions.

The third set consists of two well-known libraries for parsing xslt and xml files, *libxslt* and *libxml2*, respectively. *libxslt* has been used in [40] and *libxml2* has been used to evaluate many coverage-guided fuzzing techniques [14, 25, 33].

The criteria for selecting these three sets of programs is to demonstrate that if there are restrictive branch conditions in a program, existing fuzzing techniques have difficulty exploring program behaviors, and our proposed rare path analysis can help existing fuzzers to increase code coverage without modifying them.

We use baseline random seeds for these programs following the approaches used in prior works [25, 27, 32]. For example, when fuzzing *libxml2* or *libxslt*, as a random seed we use structured xml and xslt content respectively as the random input.

As we focus on edge coverage in our experimental evaluation, we set loop bound as 1 when analyzing rare paths in these programs.

We experimentally evaluate based on the following research questions:

RQ1. Can our rare path analysis (without using any fuzzer) generate inputs that AFL++ can not?
RQ2. Can we improve mutation-based fuzzing effectiveness using the seed set we generate from our rare path analysis?
RQ3. Can rare path-guided fuzzing achieve better coverage compared to sampling-based hybrid fuzzers?
RQ4. Can we improve the effectiveness of rare path analysis using II-paths?

Note that, similar to prior work on fuzzing (and testing in general), we rely on the hypothesis that the ability to find bugs is directly related to increasing code coverage [10]. We do not test this hypothesis in this paper rather we focus to help fuzzers by generating inputs for rare program paths (as most fuzzers struggle to exercise such paths [25, 38]), and we evaluate this by measuring the increase in coverage. Moreover, in our experimental evaluation, we focus on fuzzers like FairFuzz [25] and DigFuzz [47]. The goal of these fuzzers is to increase code coverage by exploring rare program paths which also helps these fuzzers to find more bugs. For example, FairFuzz was not experimentally evaluated for finding bugs, it was only evaluated for finding rare branches and increasing code coverage.

7.1 Experimental Setup

We run our experiments on a virtual box equipped with an Intel Core i7-8750H CPU at 2.20GHz and 16 GB of RAM running Ubuntu Linux 18.04.3 LTS. We use dockers for AFL++ [4] and FairFuzz [5] to run all the fuzzing experiments. The Test-Comp benchmark assumes that programs consume inputs via VERIFIER_nondet function calls. We redefine these functions in a wrapper so that programs can be fed with input from a file. For the first set of programs, we fuzz each program for 10 minutes and for the other two sets of programs, we fuzz each program for 24 hours. To compensate for nondeterminism we test each program at least three times and report the maximum number of edges covered. We set the upper limit for our rare path guidance technique (branch selectivity computation, rare path identification, and seed generation) to 25% of the total time (2.5 minutes for the first set and 6 hours for the second and third set) and use the remaining 75% time for fuzzing (7.5 minutes and 18 hours respectively) with the seed set generated by our analysis. We set the path depth limit to 60 for our rare path analysis. In this paper we do not experimentally evaluate the trade-off between rare path exploration and fuzzing time, however, it might be an interesting direction to explore in the future. After collecting the rare paths, we provide all the inputs from the feasible rare paths (filtered by path-guided concolic execution) to the fuzzer as the seed set.

7.2 Experimental Results

To answer RQ1, RQ2 and RQ3, we use II-Path based analysis and algorithm IIP-GCE to generate rare seeds.

7.2.1 RQ1: Effectiveness of Rare Path Analysis to Generate Rare Inputs.

Our experimental results show that the proposed rare path analysis and path-guided concolic execution generates inputs in 6 hours (without running AFL++) which AFL++ itself cannot generate in 24 hours by mutating inputs. Our results in detail are as follows.

seq-mthreaded. For 60 out of 62 programs in this set of programs, our rare path analysis generates inputs that cannot be generated by AFL++.

tinyC. Rare path analysis generates inputs containing *if-else* structure. AFL++ generates *if* structure by mutating inputs but cannot generate the *if-else* structure.

inih. AFL++ generates inputs generated by rare path analysis as the input sequence is trivial containing opening and closing brackets, and key-value pairs separated by a colon (:) or an equal sign (=).

calculator. Rare path analysis generates inputs containing keywords such as *arcsin*, *arccos*, and *arctan* but AFL++ cannot generate these keywords even after running for 24 hours.

cJSON. AFL++ generates inputs containing basic JSON structure with left and right braces, colon (:), and quotations(""). Rare path analysis generates inputs containing keywords such as *false*, *true*, and *null* that AFL++ cannot generate.

libxslt. We provide XSLT file containing opening and closing tag for `stylesheet` to AFL++. However, running AFL++ for 24 hours, it cannot generate inputs containing keywords to explore deeper functionalities. Rare path analysis generates inputs containing keywords: `attribute-set`, `preserve-space`, and `decimal-format`.

libxml2. To explore deeper paths in *libxml2*, a xml file needs to contain keywords like DOCTYPE, ATTLIST, ENTITY, NOTATION etc. Running AFL++ for 24 hours, it generates inputs containing structures like DOCTYPE and ATTLIST. Rare path analysis generates inputs containing not only DOCTYPE and ATTLIST but also ENTITY and NOTATION.

7.2.2 *RQ2: Effectiveness of Rare Path Analysis to Improve Mutation-based Fuzzing Effectiveness.*

Our experimental results on the first set of programs are shown in Table 3. Rare path analysis achieves better coverage compared to both AFL++ and FairFuzz. For 60 out of 62 programs, AFL++ with rare seed achieves better coverage compared to AFL++ with a random seed. For 48 out of 62 programs, FairFuzz with rare seed achieves better coverage compared to FairFuzz with a random seed. We see an average coverage improvement of 9.27% over AFL++ and 5.03% over FairFuzz.

Experimentally evaluating on second and third sets of programs (as shown in Fig. 5 and Table 2) we see coverage improvement over AFL++ for 5 out of 6 of the benchmarks. We do not see a lot of improvement for *calculator* (1.13%). We generate rare inputs: *arcsin*, *arccos*, and *arctan* and then mutating these rare inputs AFL++ generates 3 more inputs: *asin*, *acos*, and *atan*. However, there are not many program paths passing through these rare branches. Only 13 additional edges are covered and hence an improvement of 1.33% is achieved. For *tinyC* and *cJSON*, we see an improvement of 6.47% (13 additional edges) and 4.19% (25 additional edges), respectively. For *libxslt*, our rare path guidance helps AFL++ to cover 162 additional edges (18.86% coverage improvement). For *libxml2*, we achieve the maximum coverage improvement of 1170 additional edges (20.35%). This indicates that for larger programs with lots of program paths, if restrictive branches in the program can be passed through, existing mutation-based fuzzers can achieve significantly more code coverage.

We then experimentally evaluate using FairFuzz [25] and see similar results as AFL++. For 5 out of 6 cases, we see improvement, 0.51% for *calculator* 0.94% for *tinyC*, 4.14% for *cJSON*, 31.86% for *libxslt*, and 18.29% for *libxml2* (shown in Fig. 5 and Table 2). Moreover, for *cJSON*, *libxslt* and *libxml2*, our rare path analysis can generate inputs that FairFuzz cannot. These results indicate that FairFuzz (which uses branch hit counts to identify rare branches) can not generate rare inputs that our analysis can.

From our experimental evaluation, we also see that rare path guided FairFuzz performs the best (1.33%, 5.36%, 7.46%, 22.82% and 58.00% more coverage than AFL++ for *calculator*, *cJSON*, *tinyC*, *libxslt* and *libxml2* respectively). These results indicate that seed generation using lightweight quantitative symbolic analysis and input generation using mutation techniques is complementary to each other and a combination of these techniques is one of the promising future directions to focus on.

7.2.3 *RQ3: Comparison to Hybrid Fuzzer.*

To answer RQ3, we evaluate our rare path analysis on top of the hybrid fuzzing technique, DigFuzz [47]. DigFuzz [47] identifies the hardest paths to explore for AFL using the samples collected using AFL and then uses symbolic execution tool angr [39] to solve constraints for the hardest paths. However, DigFuzz is not publicly available. We contacted the authors of DigFuzz but could not get access to the implementation. Hence, we implement the technique in DigFuzz using AFL++ and QSym [46]. In our evaluation we use an unoptimized binary for fuzzing (to associate branch flip in concolic execution with hitcount collected in fuzzing). We experiment on the 3 largest benchmarks, *cJSON*, *libxslt* and *libxml2*. Results from our experimental evaluation (Table 4) show that rare path guided DigFuzz achieves better coverage compared to DigFuzz, 66.86% improvement for *cJSON*, 2.18% improvement for *libxslt* and 30.22% improvement for *libxml2*. There are multiple reasons behind our implementation of DigFuzz not being able to achieve better coverage compared to AFL++ and FairFuzz: 1) building the execution tree takes hours for larger programs like *libxml2* as the tree grows exponentially over time, 2) concolic execution fails to generate inputs for a lot of paths and hence generates very few inputs to guide AFL++ and 3) DigFuzz attempts to solve branches that are not dependent on the inputs rather used for sanity check of the program. These findings are aligned with the findings of DigFuzz for larger programs [47]. Nonetheless, our experiments on DigFuzz still demonstrate that rare path analysis improves the effectiveness of DigFuzz..

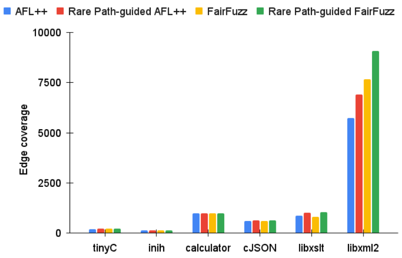


Figure 5: Coverage comparison between AFL++, rare-path guided AFL++, FairFuzz and rare-path guided FairFuzz

Table 2: Percentages of coverage improvement for rare path-guided fuzzing over AFL++, FairFuzz

Benchmarks	Number of lines	% coverage improvement over	
		AFL++	FairFuzz
tinyC	190	6.47%	0.94%
inih	243	0.00%	0.00%
calculator	1312	1.33%	0.51%
cJSON	3845	4.19%	4.14%
libxslt	33371	18.86%	31.86%
libxml2	186116	20.35%	18.29%

7.2.4 *RQ4: Effectiveness of II-path to Improve Efficiency of Rare Path Analysis.*

To answer RQ4, we guide fuzzers using our rare path analysis based on intra-paths, inter-paths and II-paths. Our experimental evaluation shows that II-paths based analysis can generate more or same number of rare inputs in comparison to both intra- and inter-paths based analysis. Experimental results for *cJSON*, *libxslt* and *libxml2* are shown in (Fig. 6). There is no improvement using intra paths for *cJSON* as it cannot generate any new inputs. However, inter paths based analysis can generate new inputs and hence improvement of 4.19%. II-paths based analysis can generate new inputs compared to intra-paths but not compared to inter-paths. For *libxslt*, there is no improvement using intra-paths as

Table 3: Coverage improvement for rare path-guided fuzzing over AFL++ and FairFuzz on seq-mthreaded programs

Program	LOC	AFL++	Rare AFL++	coverage increase	Fair Fuzz	Rare FairFuzz	coverage increase
floodmax.3.1	536	223	250	12.11%	203	223	9.68%
floodmax.3.2	559	286	307	7.34%	262	282	7.50%
floodmax.3.3	559	270	310	14.81%	282	282	0.00%
floodmax.3.4	559	280	309	10.36%	282	282	0.00%
floodmax.3	559	285	310	8.77%	275	282	2.38%
floodmax.4.1	1204	323	364	12.69%	288	328	13.64%
floodmax.4.2	1252	680	739	8.68%	714	754	5.50%
floodmax.4.3	1252	698	738	5.73%	708	747	5.56%
floodmax.4.4	1252	663	736	11.01%	695	747	7.55%
floodmax.4	1252	681	739	8.52%	708	747	5.56%
floodmax.5.1	2762	428	500	16.82%	400	452	13.11%
floodmax.5.2	2842	1937	2144	10.69%	1815	1887	3.97%
floodmax.5.3	2842	2076	2148	3.47%	1835	1914	4.29%
floodmax.5.4	2842	1987	2152	8.30%	1868	1907	2.11%
floodmax.5	2842	2067	2150	4.02%	1828	1914	4.66%
lcr-var.3.1	315	186	194	4.30%	177	177	0.00%
lcr-var.3.2	310	186	190	2.15%	170	170	0.00%
lcr-var.3	313	185	193	4.32%	177	177	0.00%
lcr-var.4.1	395	223	231	3.59%	210	216	3.12%
lcr-var.4.2	390	217	227	4.61%	203	210	3.23%
lcr-var.4	393	222	231	4.05%	210	216	3.12%
lcr-var.5.1	482	250	267	6.80%	249	256	2.63%
lcr-var.5.2	477	235	264	12.34%	236	249	5.56%
lcr-var.5	480	244	269	10.25%	242	256	5.41%
lcr-var.6.1	570	273	307	12.45%	275	301	9.52%
lcr-var.6.2	565	261	300	14.94%	269	295	9.76%
lcr-var.6	568	271	303	11.81%	275	301	9.52%
lcr.3.1	286	170	170	0.00%	144	144	0.00%
lcr.3	284	170	170	0.00%	144	144	0.00%
lcr.4.1	355	181	191	5.52%	170	170	0.00%
lcr.4	353	179	191	6.70%	164	170	3.66%
lcr.5.1	431	204	220	7.84%	197	203	3.05%
lcr.5	429	197	221	12.18%	197	203	3.05%
lcr.6.1	508	215	252	17.21%	216	236	9.26%
lcr.6	506	219	252	15.07%	223	236	5.83%
lcr.7.1	589	242	280	15.70%	236	262	11.02%
lcr.7	587	240	282	17.50%	236	262	11.02%
lcr.8.1	677	265	316	19.25%	275	301	9.45%
lcr.8	675	272	301	10.66%	256	295	15.23%
opt-f-max.3.2	604	304	329	8.22%	315	321	1.90%
opt-f-max.3.3	604	300	330	10.00%	315	321	1.90%
opt-f-max.3.4	604	300	329	9.67%	315	315	0.00%
opt-f-max.3	604	302	331	9.60%	315	321	1.90%
opt-f-max.4.1	1272	330	389	17.88%	334	374	11.98%
opt-f-max.4.2	1320	688	760	10.47%	754	800	6.10%
opt-f-max.4.3	1320	729	769	5.49%	754	793	5.17%
opt-f-max.4.4	1320	701	768	9.56%	747	800	7.10%
opt-f-max.4	1320	722	767	6.23%	760	800	5.23%
opt-f-max.5.1	2862	471	542	15.07%	452	518	14.55%
opt-f-max.5.2	2942	2082	2187	5.04%	1927	1979	2.71%
opt-f-max.5.3	2942	1955	2180	11.51%	1914	1973	3.10%
opt-f-max.5.4	2942	2031	2176	7.14%	1894	1979	4.49%
opt-f-max.5	2942	2003	2191	9.39%	1881	1979	5.23%
Standby.1	623	197	215	9.14%	170	190	11.54%
Standby.4.1	615	188	214	13.83%	164	184	12.00%
Standby.4.2	623	194	219	12.89%	190	190	0.00%
Standby.5	619	191	207	8.38%	190	190	0.00%
Standby	623	189	216	14.29%	190	190	0.00%
Triplicated.1	539	192	197	2.60%	170	170	0.00%
Triplicated.2	535	196	199	1.53%	170	170	0.00%
Triplicated	543	200	203	1.50%	177	177	0.00%

it can not generate new inputs. Using inter-paths, there is improvement of 9.08% as new inputs are generated containing keywords `preserve-space` and `decimal-format`. However, using II-paths, improvement of 17.93% is achieved as inputs containing keyword (`attribute-set`) is generated, not generated by both intra- and inter- paths. For *libxml2*, identifying rare paths based on intra-paths can generate an input containing the specific value `DOCTYPE` and hence, there is coverage improvement. Intra-paths based analysis can generate `DOCTYPE` as the branch conditions comparing to this specific value are present in the initial starting procedure. There is no improvement using inter-paths, rather coverage is reduced as inter-paths can not find any rare inputs. Inter-paths based analysis explores program paths deep inside the nested procedures and most of these paths are not rare paths and due to the exponential increase in the number of program paths, analysis time is increased. On the other side, II-paths can generate inputs containing specific values `DOCTYPE`, `ATTLIST`, `ENTITY` and `NOTATION`. These inputs help to achieve better coverage compared to both intra- and inter- paths based analysis.

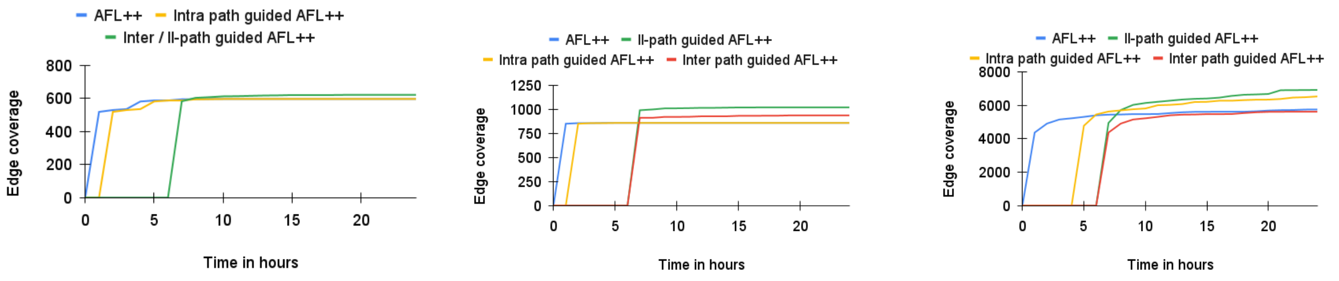
Table 4: Percentages of coverage improvement for rare path-guided fuzzing over DigFuzz

Benchmarks	DigFuzz	Rare Path-guided DigFuzz	% coverage improvement
cJSON	344	574	66.86%
libxslt	719	735	2.18%
libxml2	3297	4270	30.22%

8 RELATED WORK

Mutation-based coverage guided Fuzzers. AFL [33] is a well-known mutation-based coverage guided fuzzer. AFL++ [19] is the latest version of AFL. In this work, we use default version of AFL++ which uses power schedule of AFLFast [14]. MOPT [31] focuses on mutation scheduling by providing different probabilities to the mutation operators. LAF-INTEL [1] focuses on bypassing hard multibyte comparisons, by splitting them into multiple single-byte comparison. REDQUEEN [8] focuses on bypassing Input-To-State (I2S) defined comparisons. Steelix[28] performs static analysis and extra instrumentation to produce inputs satisfying multi-byte comparisons. VUzzer [36] identifies input positions used in the comparison and immediate values using a Markov Chain model and decides which parts of the program should be targeted. FairFuzz [25] identifies the rare branches in the program based on the hitcounts of branches. If a rare branch is identified by FairFuzz, it applies input mutation masking. In this paper, we focus on identifying rare program paths. We neither use a fuzzer to identify rare paths, nor modify mutation strategies inside the fuzzer. We show that we can improve the effectiveness of state of the art fuzzers without making any changes to the internals of fuzzers.

Symbolic execution guided fuzzers. Hybrid fuzzing techniques [30, 38, 47] use symbolic execution and constraint solvers to generate inputs to pass complex checks in the program. Driller [38] uses selected symbolic execution when fuzzer can not cover new branches. DigFuzz [47] uses the fuzzer itself to statistically identify hardest paths for the fuzzer to explore and then uses symbolic execution to solve path constraints for the hardest paths. DeepFuzzer [30] uses lightweight symbolic execution to pass initial complex checks and then it relies on seed selection and mutation techniques. In this work, we do not use the path samples from fuzzer to identify rare



(a) cJSON **(b) libxslt** **(c) libxml2**

Figure 6: Coverage improvement comparison between different types of path-guided fuzzing. II-paths can generate more number of rare inputs compared to both intra and inter paths within a given amount of time and hence highest edge coverage is achieved by II-path guided fuzzing.

paths, rather we statically analyze programs. Moreover, we do not symbolically execute the whole program, instead guide symbolic execution using the rare paths.

Grammar-based Fuzzers. Grammar-based fuzzing techniques generate well-formed inputs based on a user provided grammar [22, 44]. These fuzzing techniques mutate inputs using the derivative rules in the grammar. As a result, the mutated input is also guaranteed to be well-formed [29]. Grammar-based fuzzers are very effective to fuzz programs that are heavily dependent on structured inputs [22, 42]. However, grammar-based fuzzers require application specific knowledge of the program under test. There are several fuzzers [15, 24, 32, 40, 43] focus specifically on structured inputs. Our technique does not require any knowledge about the program and it is fully automated. We neither need to provide an input grammar, nor feed inputs to the parser [32, 42] or collect large data samples [40] like techniques that specialize on structured inputs.

Seed generation for fuzzers. There are fuzzing techniques that focus on seed selection and seed prioritization to improve fuzzing efficiency [23, 35, 45]. SpotFuzz [35] identifies invalid execution and time consuming edges as hot spots based on hitcounts of different inputs on the edges. SLF [45] is a technique which focuses on valid seed input generation. It performs sophisticated input mutation to get through the validity checks. [23] systematically investigates and evaluates the affect of seed selection on fuzzer's ability to find bugs and demonstrates that fuzzing outcomes vary depending on the initial seeds used. In this work, we also demonstrate that rare inputs as initial seeds bootstraps the fuzzer. However, we focus on generating seeds that can execute rare paths.

Static program analysis for fuzzing. A large number of fuzzing techniques [8, 11, 20, 28, 32, 36] use static program analysis techniques to guide fuzzers. Most of these techniques use either control flow analysis or taint analysis. In this work, we also use control flow analysis and dependency analysis to identify rare paths. However we introduce a novel technique we call rare path analysis and a new kind of control flow paths (II-paths). Although different, our definition of II-paths is inspired by the control flow directed concolic search techniques provided in [16].

9 CONCLUSIONS

In this paper, we provide techniques to identify rare program paths that are difficult for a fuzzer to explore generating random inputs. To identify the rare paths, we use lightweight static analysis. We use the identified rare paths to guide a concolic execution tool to generate inputs that can execute these rare paths. Finally, we provide these inputs as the initial seed set to the fuzzer. From our experimental evaluation on 3 different set of benchmarks, we find that our approach generates inputs that a fuzzer cannot generate by mutations. Inputs generated by our analysis guide existing fuzzers to achieve better coverage compared to an initial random seed.

REFERENCES

[1] 2006. laf-intel. https://lafintel.wordpress.com/. Accessed: 2018-08-21.
[2] 2022. Calculator. https://github.com/btmills/calculator.Accessed: 2022-08-21.
[3] 2022. CodeQL. https://codeql.github.com.Accessed: 2022-08-21.
[4] 2022. Docker for AFL++. https://hub.docker.com/r/aflplusplus/aflplusplus.Accessed: 2022-08-21.
[5] 2022. Docker for FairFuzz. https://hub.docker.com/r/zjuchenyuan/fairfuzz.Accessed: 2022-08-21.
[6] 2023. SV-Benchmark:seq-mthreded. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c/seq-mthreaded.Accessed: 2023-02-15.
[7] Frances E. Allen. 1970. Control Flow Analysis. *SIGPLAN Not.* 5, 7 (jul 1970), 1–19. https://doi.org/10.1145/390013.808479
[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. https://doi.org/10.14722/ndss.2019.23371
[9] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. 255–272. https://doi.org/10.1007/978-3-319-21690-4_15
[10] Thomas Bach, Artur Andrzejak, Ralf Pannemans, and David Lo. 2017. The Impact of Coverage on Bug Density in a Large Industrial Software Project. https://doi.org/10.1109/ESEM.2017.44
[11] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2012. A Taint Based Approach for Smart Fuzzing. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012* (04 2012). https://doi.org/10.1109/ICST.2012.182
[12] Dirk Beyer. 2021. *Software Verification: 10th Comparative Evaluation (SV-COMP 2021)*. 401–422. https://doi.org/10.1007/978-3-030-72013-1_24
[13] Dirk Beyer. 2022. *Advances in Automatic Software Testing: Test-Comp 2022.* 321–335. https://doi.org/10.1007/978-3-030-99429-7_18
[14] Marcel Bohme, Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* PP (12 2017), 1–1. https://doi.org/10.1109/TSE.2017.2785841
[15] Sergey Bratus, Axel Hansen, and Anna Shubina. 2008. LZfuzz: a fast compression-based fuzzer for poorly documented protocols. (2008).
[16] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 443–446. https://doi.org/10.1109/ASE.2008.69

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[18] Antonio Filieri, Corina Păsăreanu, Willem Visser, and Jaco Geldenhuys. 2014. Statistical symbolic execution with informed sampling. 437–448. https://doi.org/10.1145/2635868.2635899

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[20] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2577–2594.

[21] Jaco Geldenhuys, Matthew Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. *2012 International Symposium on Software Testing and Analysis, ISSTA 2012 - Proceedings* (07 2012). https://doi.org/10.1145/2338965.2336773

[22] Patrice Godefroid, Adam Kiezun, and Michael Levin. 2008. Grammar-based Whitebox Fuzzing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 43, 206–215. https://doi.org/10.1145/1379022.1375607

[23] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony Hosking. 2021. Seed selection for successful fuzzing. 230–243. https://doi.org/10.1145/3460319.3464795

[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458.

[25] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. 475–485. https://doi.org/10.1145/3238147.3238176

[26] Caroline Lemieux and Koushik Sen. 2021. FairFuzz-TC: a fuzzer targeting rare branches. *International Journal on Software Tools for Technology Transfer* 23, 6 (01 Dec 2021), 863–866. https://doi.org/10.1007/s10009-020-00569-w

[27] Caroline Lemieux and Koushik Sen. 2021. FairFuzz-TC: a fuzzer targeting rare branches. *International Journal on Software Tools for Technology Transfer* 23, 6 (01 Dec 2021), 863–866. https://doi.org/10.1007/s10009-020-00569-w

[28] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[29] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476

[30] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. 2021. DeepFuzzer: Accelerated Deep Greybox Fuzzing. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (2021), 2675–2688. https://doi.org/10.1109/TDSC.2019.2961339

[31] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[32] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-Directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 548–560. https://doi.org/10.1145/3314221.3314651

[33] Michał Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[34] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–228.

[35] Haibo Pang, Jie Jian, Yan Zhuang, Yingyun Ye, and Zhanbo Li. 2021. SpotFuzz: Fuzzing Based on Program Hot-Spots. *Electronics* 10 (12 2021), 3142. https://doi.org/10.3390/electronics10243142

[36] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. https://doi.org/10.14722/ndss.2017.23404

[37] Seemanta Saha, Mara Downing, Tegan Brennan, and Tevfik Bultan. 2022. PREACH: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1706–1717. https://doi.org/10.1145/3510003.3510227

[38] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. https://doi.org/10.14722/ndss.2016.23368

[39] Fish Wang and Yan Shoshitaishvili. 2017. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. 8–9. https://doi.org/10.1109/SecDev.2017.14

[40] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23

[41] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. 2022. Evaluating and Improving Neural Program-Smoothing-based Fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 847–858. https://doi.org/10.1145/3510003.3510089

[42] Jingbo Yan, Yuqing Zhang, and Dingning Yang. 2013. Structurized grammar-based fuzz testing for programs with highly structured inputs. *Security and Communication Networks* 6 (11 2013). https://doi.org/10.1002/sec.714

[43] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[44] Hyunguk Yoo and Taeshik Shon. 2016. Grammar-based adaptive fuzzing: Evaluation on SCADA modbus protocol. 557–563. https://doi.org/10.1109/SmartGridComm.2016.7778820

[45] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without Valid Seed Inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 712–723. https://doi.org/10.1109/ICSE.2019.00080

[46] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[47] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing.. In *NDSS*. https://doi.org/10.14722/ndss.2019.23504

Received 2023-02-16; accepted 2023-05-03