Enhancing Coverage-Guided Fuzzing via Phantom Program

Mingyuan Wu
Southern University of Science and Technology
Shenzhen, China
The University of Hong Kong
Hong Kong, China
11849319@mail.sustech.edu.cn

Kunqiu Chen
Qi Luo
Jiahong Xiang
11911626@mail.sustech.edu.cn
12232440@mail.sustech.edu.cn
11812613@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Junjie Chen
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Heming Cui
The University of Hong Kong
Hong Kong, China
heming@cs.hku.hk

Ji Qi
The University of Hong Kong
Hong Kong, China
jqi@cs.hku.hk

Yuqun Zhang†
Southern University of Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

ABSTRACT
For coverage-guided fuzzers, many of their adopted seeds are usually ineffective by exploring limited program states since essentially all their executions have to abide by rigorous dependencies between program branches while only limited seeds are capable of accessing such dependencies. Moreover, even when iteratively executing such limited seeds, the fuzzers have to repeatedly access the covered program states before uncovering new states. Such facts indicate that exploration power on program states of seeds has not been sufficiently leveraged by the existing coverage-guided fuzzing strategies. To tackle these issues, we propose a coverage-guided fuzzer, namely MirageFuzz, to mitigate the dependencies between program branches when executing seeds for enhancing their exploration power on program states. Specifically, MirageFuzz first creates a "phantom" program of the target program by reducing its dependencies corresponding to conditional statements while retaining their original semantics. Accordingly, MirageFuzz performs dual fuzzing, i.e., the source fuzzing to fuzz the original program and the phantom fuzzing to fuzz the phantom program simultaneously. Then, MirageFuzz generates a new seed for the source fuzzing via a taint-based mutation mechanism, i.e., updating the target conditional statement of a given seed from the source fuzzing with its corresponding condition value derived by the phantom fuzzing. To evaluate the effectiveness of MirageFuzz, we build a benchmark suite with 18 projects commonly adopted by recent fuzzing papers, and select nine open-source fuzzers as baselines for performance comparison with MirageFuzz. The experiment results suggest that MirageFuzz outperforms our baseline fuzzers from 13.42% to 77.96% averagely. Furthermore, MirageFuzz exposes 29 previously unknown bugs where 7 of them have been confirmed and 6 have been fixed by the corresponding developers.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging.

KEYWORDS
Fuzzing, Coverage Guidance, Phantom Program

ACM Reference Format:

1 INTRODUCTION
Fuzzing [44] refers to automatically generating invalid, unexpected, or random test inputs (i.e., seeds) to expose unexpected program behaviors, e.g., crashes and memory leaks, which can be further analyzed to detect vulnerabilities/bugs of target programs. In particular, many existing fuzzers [18, 28, 42, 47, 78] have widely adopted code coverage as guidance of their fuzzing strategies to advance bug/vulnerability exposure. Typically, based on an initial collection of seeds, a coverage-guided fuzzer develops its fuzzing strategy to iteratively generate new seeds (often via mutation) for increasing/optimizing code coverage.

Albeit many coverage-guided fuzzers have been shown effective in terms of code coverage and bug exposure [23, 39, 45, 72], their
coverage-guided strategies are still somewhat restricted to hinder their further performance improvement. In particular, the existing coverage-guided fuzzing strategies typically require complete execution on each seed, i.e., exploring program states bounded by rigorous dependencies between program branches (referred to as program dependencies in the rest of the paper for simplicity). It has been widely shown that in this way, many seeds are executed to only result in the limited state exploration of target programs [42, 63, 79, 80], indicating that a large number of such seeds are ineffective in exposing new program states. Furthermore, even for the limited number of seeds which can effectively explore program states, their iterative executions are still subject to rigorous program dependencies, i.e., incrementally exploring program states in order. As a result, the fuzzers have to repeatedly access the covered program states before uncovering new program states under each iterative execution. Such facts indicate that the seed-wise exploration power on program states has not been sufficiently leveraged by the existing coverage-guided fuzzing strategies.

In this paper, we attempt to tackle the aforementioned limitations of the seed-wise exploration power for the existing coverage-guided fuzzing strategies. Our key insight is that instead of only using a limited number of effective seeds for incrementally exploring program states under iterative executions, we seek to exploit more effective seeds as well as the exploration on separate program states by reducing their inter-dependencies so as to enhance the efficacy of coverage-guided fuzzing. Accordingly, we propose MirageFuzz, the first fuzzer which attempts to mitigate the rigorous compliance with all inter-dependencies between program states when executing coverage-guided fuzzing strategies for enhancing the exploration power of all seeds. To this end, for a given target program, we first derive its control flow graph and identify the conditional instruction in each basic block and all the instructions affecting it in the intermediate representation (IR) level [41, 59]. Then we relocate such instructions to their farthest dominator while preserving the original semantics of the conditional instruction, i.e., reducing program dependencies, as forming a "phantom" program. Next, MirageFuzz performs dual fuzzing, i.e., fuzzing the original program and its phantom program simultaneously, namely source fuzzing and phantom fuzzing. More specifically, after the source fuzzing upon a given seed S, we collect the unexplored program branches adjoining the explored program states and search for any seed generated by the phantom fuzzing which can be executed to explore any of such branches. If such a seed S′ exists, we then identify the byte offset of S′ corresponding to the conditional instruction of the unexplored branch via taint analysis [60] and further update it using the corresponding condition value derived by S′ to form a new seed for further source fuzzing. Eventually, executing the resulting new seed can advance the exploration of the program states bounded by the corresponding conditional instruction and thus enhances the seed effectiveness on exploring program states.

To evaluate the effectiveness of MirageFuzz, we first collected 18 real-world projects which were frequently adopted in recent fuzzing research as our benchmark suite. We further collected nine open-source coverage-guided fuzzers as our baselines for performance comparison with MirageFuzz. Our evaluation results suggest that MirageFuzz outperformed the baseline fuzzers significantly by 13.42% to 77.96% on average in terms of the edge coverage. Moreover, MirageFuzz exposed 29 previously unknown bugs where 7 of them have been confirmed and 6 have been fixed by the corresponding developers.

In summary, our paper makes the following contributions:

• **Idea.** To the best of our knowledge, we are the first to propose the concept of phantom program which reduces program dependencies and perform dual fuzzing to synergize the source fuzzing and the phantom fuzzing to enhance coverage-guided fuzzing.

• **Technique.** We have implemented the proposed idea as an open-source practical tool, namely MirageFuzz, as released in our GitHub page [4].

• **Evaluation.** We evaluate MirageFuzz upon a real-world benchmark with 18 open-source projects compared with nine baseline fuzzers. The evaluation results indicate that MirageFuzz outperforms all baseline fuzzers averagely from 13.42% to 77.96% in terms of edge coverage. Moreover, MirageFuzz exposed 29 previously unknown bugs where 7 of which have been confirmed and 6 have been fixed by the corresponding developers.

2 MOTIVATION

In this section, we use a sample code snippet following prior work as in Figure 1a [43] to motivate MirageFuzz. Specifically, the function Origin takes a character array as input and processes it in nested branches. Note that many existing coverage-guided fuzzers [18, 28, 42, 47, 72, 78] incrementally increase code coverage under each iterative execution. Therefore, to trigger the crash on line 7 of function Origin, first, given an initial seed successfully exploring line 3, it is ideal to generate a mutant which can be executed to successfully explore line 4 under controllable effort with the resulting mutant retained as the new seed. The above operation is then repeated for the subsequent statements until line 6 can be successfully explored. However, the mutation space for each statement is essentially vast, e.g., for line 3, user[8] can be assigned with 256 possible values while it has to be 'M' only to successfully access its scope. Thus, We can derive that the chance of a seed to explore consecutive similar statements could be rather trivial, resulting in many underused seeds for fuzzing. To summarize, the exploration power on program states of a seed can be somewhat limited by applying many existing coverage-guided fuzzing strategies.

We consider the key factor limiting the effectiveness of coverage-guided fuzzing strategies is that they require the seeds to rigorously abide by the program dependencies, i.e., being thoroughly executed, until exposing a bug/vulnerability. Specifically in Figure 1a, the execution on one seed has to satisfy all the dependencies of line 7, i.e., lines 3 to 6, before exposing the relevant crash. Moreover, the program states subject to such program dependencies even have to be repeatedly accessed under iterative executions, e.g., line 3 has to be explored by all the iterative executions until exploring line 7. Therefore, in this paper, we attempt to enhance the effectiveness of coverage-guided fuzzing strategies by mitigating the rigorous compliance with all inter-dependencies between program states on performing coverage-guided fuzzing strategies. In particular, a straightforward insight is to reduce program dependencies for preventing the aforementioned executions. We can observe from Figure 1a that actually the conditional statements of lines 3 to 6...
We attempt to obtain the byte offset of the running seed. Accordingly, in this paper, we are inspired to propose a technique. Therefore, it is unnecessary to form nested dependencies among program branches. Hence, the exploration of program states which many existing fuzzers [24, 25, 39, 43, 64, 77] attempt to tackle by proposing diverse techniques, e.g., recording the auxiliary states for program exploration depth. However, due to the aforementioned limitation of the well-adopted coverage guidance, they still generate massive ineffective seeds, i.e., only limited seeds are effective to explore program states. To illustrate, that essentially is the issue we attempt to address in this paper.

3 APPROACH

Figure 2 shows the overall workflow of MirageFuzz which consists of three components. First, MirageFuzz creates a phantom program to reduce dependencies in the target program via a dependency reduction mechanism (marked as ⊙ in Figure 2, Section 3.1). Next, MirageFuzz performs dual fuzzing—the source fuzzing to fuzz the original program and the phantom fuzzing to fuzz its corresponding phantom program simultaneously (⊙, Section 3.2). Specifically, during iterative executions of the source fuzzing under a given seed, MirageFuzz obtains the unexplored program branches adjoining the explored program states and searches for whether any of them has been explored by a seed (or multiple seeds) generated from the phantom fuzzing. At last, if such a seed S' exists, we then update the corresponding branch condition of S with the value derived by S' to form a new seed for future source fuzzing (⊙, Section 3.3).

3.1 Dependency Reduction Mechanism

We first derive the control-flow graph (CFG) of the given target program and then identify all the branch instructions in the intermediate representation (IR) level [41, 59]. Next, for the conditional instruction in each basic block and all the instructions affecting it, we attempt to relocate them to their farthest dominator (in this paper, we follow prior work [3] that in a control-flow graph, a block a is a dominator of a block b if every path from the entry block to b must go through a). In this way, we essentially reduce the dependencies among program branches. Here we use the code snippets from a real-world project jhead [5] in Figure 3 with the CFG generated by LLVM dot-cfg pass [2] for illustration. In particular, Figure 3 presents a total of 8 basic blocks where T represents that the corresponding condition is evaluated as "true" and F represents otherwise. For instance, by relocating the conditional instruction bool cmp3 = !strcmp(arg, "-proc") of block ④ in its farthest dominator, i.e., entry block ①, their original dependency can thus be reduced. As a result, the execution on a seed can directly explore block ④ without exploring its original dependency with entry block ① in advance. Obviously, the chance to explore block ④ can be increased, indicating the exploration power of seeds can be increased.

Inspired by Section 2, we realize that by preserving the entry condition of each branch in the target program, we can utilize the dependency-reduced program to facilitate the exploration of new program states in the original program. However, relocating instructions can easily violate the semantics of the original program branches (i.e., the entry condition of a program branch). For instance in Figure 3, relocating instructions from block ⑤ to entry block ① can violate the original semantics of block ③. In particular, by executing the instruction sscanf(argv[++n], "%d", &P) of block ③ in entry block ① after instruction relocation, the value of variable ret (in instruction as via instruction sscanf(argv[++n], "%d", &PQ) of block ⑧ is changed since its variable n has already been updated in entry block ③. Accordingly, the semantics of condition bool cmp2 = (ret != 1) for block ⑧ would be violated. Note that while it is essential to preserve the semantics of branch conditions for correctly exploring their covered program states by executing seeds, it is unnecessary to preserve the semantics of other statements since changing them exerts no impact in accessing the updated blocks (i.e., for an operand not in a branch condition, its value is allowed to be changed after branch relocation if it is irrelevant to the operand value(s) of any branch condition).

In this paper, we propose a dependency reduction mechanism to reduce program dependencies by relocating conditional instruction and the instructions affecting it while preserving original semantics of each conditional instruction as shown in Algorithm 1. For a given CFG of the target program, we first obtain all the basic blocks (represented as b1locks) with the conditional instruction of each basic block (represented as branchCon, lines 2 to 4). Next, for each b1ock,

```
1 void origin(char *user){
2     user[4] = 'M';
3     if (user[8] == 'M') {
4         if (user[1] == 'A') {
5             if (user[2] == '2') {
6                 if (user[3] == 'E') {
7                     // crash
8                 } else {
9                     // other code
10                 } // other code
11             } else {
12             } return;
13         } return;
14     } return;

(a) The original code

1 void Phantom(char *user)
2     user[4] = 'A';
3     if (user[8] == 'M') {
4         if (user[1] == 'A') {
5             if (user[2] == '2') {
6                 if (user[3] == 'E') {
7                     // crash
8                 } else {
9                     // other code
10                 } // other code
11             } else {
12             } return;
13         } return;
14     } return;

(b) The phantom code
```

Figure 1: A motivation example code for MirageFuzz

are not related to one another, i.e., each of them can be satisfied independently (the operands of line 6 are irrelevant with line 3). Therefore, it is unnecessary to form nested dependencies among such conditional statements. Instead, we could reduce their dependencies as in function Phantom of Figure 1b where their respective executions are independent from each other. For example, the executions on all the seeds can easily access line 6 to check whether the runtime value of user[3] is ‘E’. Thus, we can infer that the chance to expose the crash in line 7 can be significantly enhanced compared with Figure 1a. Such an example can be rather inspiring for how to enhance the power of exploring program states of the mutants. Specifically, suppose a fuzzing campaign is halted upon how to enhance the power of exploring program states of the mutants. Specifically, suppose a fuzzing campaign is halted upon some conditions for correctly exploring their covered program states. However, relocating in-
we identify all the basic blocks with the instructions affecting its conditional instruction, i.e., sharing variable usage, via program slicing [71]. Note that for the function slicingBasicBlocks, we perform the program slicing which extracts the dependent instructions of branchCon by applying the use-def chains in both LLVM IR [41] and Memory SSA [9, 53] to obtain their associated blocks collectedBlocks (line 5). Furthermore, we filter out the dominators of branchCon whose semantics can be possibly violated by relocating the conditional instruction and the instructions affecting it (line 6). If there is no remaining basic block after filtering, we can infer that all the instructions can be relocated in the entry block (lines 7 to 8). Otherwise, we identify the farthest dominator in CFG to which all the instructions can be relocated without violating semantics on the conditional instruction (lines 10 to 12) and perform the instruction relocation (lines 13 to 14). After the iterative executions on all the collected conditional instructions, we obtain a phantom program for MirageFuzz (line 15). Note that for the phantom program, we only preserve semantics for conditional instructions as in the original program such that its adopted seeds can be used to advance program state exploration when fuzzing the original program (illustrated later) while the semantics of the rest instructions does not matter for building our phantom program. Accordingly in Figure 3, the conditional instructions located in ② and ⑤ cannot be relocated since ++n violates the semantics of the branch conditions where n is an operand. On the contrary, the conditional instructions in ⑥ and ④ are relocated to ⑤ and ① since the semantics of all involved branch conditions can be preserved after relocation (more details are presented in our GitHub page [4]).

### 3.2 Dual Fuzzing

Given the phantom program by applying the dependency reduction mechanism, MirageFuzz performs dual fuzzing, i.e., the source fuzzing to fuzz the original program and the phantom fuzzing to fuzz the phantom program simultaneously, under the identical initial seed corpus and execution time budget. During the source fuzzing, MirageFuzz collects the unexplored program branches adjoining the explored program states. Then, MirageFuzz searches for any seed
We derive the unexplored program edges adjoining the explored path, i.e., $A \rightarrow \forall$, $A \rightarrow \forall$, and $\forall \rightarrow \forall$, which are further collected in edgeDic for the phantom fuzzing. For each edge, MirageFuzz searches for any seed executed to explore it in the phantom fuzzing. At last, all the collected seeds from the phantom fuzzing are used to generate new seeds for future source fuzzing via the taint-based mutation mechanism.

### 3.3 Taint-based Mutation Mechanism

We develop the taint-based mutation mechanism to derive the byte offset corresponding to the conditional instruction of the given seed from the source fuzzing via taint analysis [60]. Specifically, in our adopted taint analysis, the input stream (i.e., the seed) is referred to as the sole taint source. In order to trace the tainted labels at runtime, we define taint propagation rules to map the tainted input labels and output labels (e.g., add, store, and load instructions) at a particular level of the operation hierarchy. Accordingly, given a specific branch condition, we can collect its corresponding label for its operand or the relevant byte offset of such a operand in the seed.

Algorithm 3 illustrates the details of the taint-based mutation mechanism which is initialized with the seed to be mutated by the source fuzzing (denoted as sourceSeed) and the collected seeds from the phantom fuzzing which can be executed to explore the identified unexplored edges from the source fuzzing (denoted as phantomSeeds). We first obtain all real-time explored edges (line 3). Next, for each seed in the phantomSeeds, we iterate its explored edges (lines 4 to 5). If the edge is not explored by the source fuzzing, we then derive the byte offset corresponding to the conditional instruction by taint analysis in the sourceSeed (lines 6 to 7).

In Figure 3, assume a seed is executed to explore the path $[3, 4, 5, 6, 7, 8]$ for the source fuzzing. Then we can derive the unexplored edges adjoining the explored path, i.e., $3 \rightarrow 4, 4 \rightarrow 5$, and $5 \rightarrow 6$, which are further collected in edgeDic for the phantom fuzzing. For each edge, MirageFuzz searches for any seed executed to explore it in the phantom fuzzing. At last, all the collected seeds from the phantom fuzzing are used to generate new seeds for future source fuzzing via the taint-based mutation mechanism.
in the phantom fuzzing, we then apply taint analysis again on that seed to obtain values of the involved operand corresponding to the branch condition. Accordingly, we generate a mutant by updating taintPos with the corresponding value from the seed generated by the phantom fuzzing, and then store it in the set taintSeeds (lines 9 to 11). At last, the resulting taintSeeds is used for advancing the future source fuzzing. We take the same seed $S$ exploring path \([1, 4, 5, 7, 8]\) mentioned in Section 3.2 as an example. Suppose we have another seed $S'$ generated by the phantom fuzzing which has satisfied the conditional instruction for block $6$. Next, by performing taint analysis on $S$, we identify its byte offset impacting the value of PO that determines the transition $\text{src} \rightarrow \text{tgt}$ or $\text{src} \rightarrow \text{alt}$. We further figure out that the value of PO in $S'$ is 14 via taint analysis on $S'$. Eventually, we replace the value of PO in $S$ with 14 to generate a new seed for exploring the new edge $\text{src} \rightarrow \text{tgt}$ for the source fuzzing.

4 IMPLEMENTATION

We implement MirageFuzz using C/C++. Specifically, we perform instrumentation via LLVM pass [41] to obtain runtime information of target programs. Accordingly, we build MirageFuzz via the AFL implementation. Furthermore, we modify the taint analysis library libdf3 [36] to implement the taint-based mutation mechanism.

We encounter three main challenges when implementing MirageFuzz. First, it is challenging to identify the unexplored edges adjoining the explored program states via instrumentation. Second, adapting the existing taint analysis tool for the taint-based mutation mechanism in MirageFuzz potentially leads to non-negligible engineering effort. At last, implementing phantom fuzzing tends to cause unexpected crashes which terminate the execution on the phantom program early to prevent it from exploring deep program states. We then illustrate how we address the challenges as follows.

4.1 Instrumentation

Note that in the source fuzzing, we aim at recognizing unexplored edges adjoining the explored program states by executing a seed. To this end, we insert an observation instruction ahead of a given instruction (lines 9 to 11). At last, the resulting taintSeeds is used for advancing the future source fuzzing. We take the same seed $S$ exploring path \([1, 4, 5, 7, 8]\) mentioned in Section 3.2 as an example. Suppose we have another seed $S'$ generated by the phantom fuzzing which has satisfied the conditional instruction for block $6$. Next, by performing taint analysis on $S$, we identify its byte offset impacting the value of PO that determines the transition $\text{src} \rightarrow \text{tgt}$ or $\text{src} \rightarrow \text{alt}$. We further figure out that the value of PO in $S'$ is 14 via taint analysis on $S'$. Eventually, we replace the value of PO in $S$ with 14 to generate a new seed for exploring the new edge $\text{src} \rightarrow \text{tgt}$ for the source fuzzing.

4.2 Dynamic Taint Analysis

We adopt libdf3 [36], a stable and efficient binary-level dynamic taint analysis framework adopted by many existing works [23, 58, 77], to implement the taint-based mutation mechanism. Although libdf3 implemented the taint propagation rules for 146 instructions, their default taint propagation rules still cannot cover our required instructions, e.g., bsTap (reversing the byte order of a register) and shl (shifting the bits of a register to the left). We also analyze that multiple taint labels of instructions movzx and movsx can cause “over-taint” issues, leading to inefficient taint tracking. To tackle these issues, we define our own taint propagation rules to cope with 11 new instructions and revoke the redundant taint labels for libdf3 to improve the taint-based mutation mechanism.

4.3 Crash Handling in Phantom Fuzzing

Generating the phantom program can inevitably devastate many dependencies of the original program, incurring crashes which potentially prevent the phantom fuzzing from exploring sufficient states of the phantom program. To address this issue, we design a “try-catch” mechanism to bypass these unexpected crashes. More specifically, we first capture all crash-related system signals and design their corresponding handler. Next, we obtain the runtime program counter [56] value, and increase it with the length of the real-time crash-triggered instruction to bypass it. As a result, phantom fuzzing can proceed to explore program states instead of being halted by the unexpected crashes.

With the solutions above, MirageFuzz is made scalable since it can be directly adopted upon any projects built upon LLVM-based compiler (e.g., clang [40]) without any additional adaptation effort.

5 EVALUATION

In this section, we conduct a set of experiments to evaluate the effectiveness of MirageFuzz upon 18 benchmark programs compared with nine baseline fuzzers. In particular, we attempt to answer the following research questions:

- RQ1: Is MirageFuzz effective compared with the baseline fuzzers?
- RQ2: Is each component of MirageFuzz effective in terms of ablation study?

We also report and analyze the bugs on our benchmark suite exposed by MirageFuzz. Note that all source code of MirageFuzz and the evaluation details are presented in our GitHub pages [1, 4].

5.1 Baseline Fuzzers and Benchmark

Baseline fuzzers. To collect the baseline fuzzers for performance comparison with MirageFuzz, we determine to first select the coverage-guided fuzzers recently published in prestigious software engineering and security conferences, e.g., ICSE, FSE, S&P, and CCS. Next, we filter the selected fuzzers based on their source code availability and the feasibility of their execution environments. Eventually, we collect a total of nine fuzzers to form our baselines. More specifically, we select six coverage-guided fuzzers, i.e., the latest versions of AFL [78], AFL++ [28], LaFIntel [7], HavocMAB [72], MOPT [47] and FairFuzz [42]. Moreover, we also adopt three recent fuzzers with constraint solvers as our baselines, i.e., Angora [23], MEUZZ [25] and QSYM [77], to further compare the performance of our insight which enhances the exploration power of seeds without leveraging the power of the constraint solver and the constraint-solving-based fuzzers on their well-performed benchmarks.

Benchmark. Following multiple prior works [23, 42, 47, 72, 77], we first construct our benchmark suite by collecting the projects commonly adopted by the fuzzers recently published in the aforementioned top software engineering and security conferences. Next, we also include 6 projects from FuzzBench [50] in our benchmark suite. As a result, our benchmark suite is formed by 18 frequently used projects with their latest versions. We also present the statistics of our adopted benchmarks in our GitHub page [1].
5.2 Environment Setup
Our evaluations are performed on a server with 64-core 2.80GHz Intel(R) Xeon(R) Gold 6342 CPUs and 64 GiB RAM running on 64-bit Linux version 4.15.0-172-generic Ubuntu 18.04.

Following many prior work [7, 28, 42, 47, 72, 77, 78], we set the total execution time budget to 24 hours. Meanwhile, all our evaluation results are averaged out of 10 runs. Furthermore, we follow the seed selection strategy in prior work [33, 37, 42, 70] to construct the initial seed corpus for each benchmark program from either its corresponding AFL seed collection or its own test suite.

In this paper, we adopt edge coverage to represent code coverage, as all our studied baseline fuzzers [7, 28, 42, 47, 72, 77, 78]. Here an edge refers to a conditional jump between two basic blocks in the program control flow. Note that since MirageFuzz enables two instances in dual fuzzing, for fair performance comparison, we evaluate all our baseline fuzzers in a parallel fuzzing manner, i.e., enabling one additional instance which shares the same seed corpus during the fuzzing campaign for all the baseline fuzzers (except QSYM and MEUZZ which enable three processes sharing the same seed queue by default [25, 77]).

5.3 Result Analysis

5.3.1 RQ1: the effectiveness of MirageFuzz. Table 1 presents the edge coverage results of our studied fuzzers upon our benchmark suite. Noticing that MEUZZ requires additional computation resource to analyze the target program for fuzzing, we mark a benchmark as N/A when MEUZZ fails to complete its execution after consuming all memory resource (e.g., objcopy). Overall, we can observe that MirageFuzz outperforms all other fuzzers significantly. In particular, MirageFuzz explores 5773 edges on average, which is 13.42% more than the top-performing baseline fuzzer QSYM (5090 explored edges) and 77.96% more than the worst-performing baseline fuzzer Laflint (3244 explored edges) in our study. Additionally, MirageFuzz consistently outperforms all the baseline fuzzers upon each benchmark program. To illustrate the significance of the performance, we also adopt the Mann-Whitney U test [48] in our evaluation. We can observe that in Table 1 where the p-value of MirageFuzz comparing with other studied fuzzers in terms of the average edge coverage are all far below 0.05, which indicates that MirageFuzz outperforms all selected fuzzers significantly ($p < 0.05$). Furthermore, Figure 4 presents the edge coverage trends of all our studied fuzzers upon each benchmark program within the 24-hour execution. We can observe that MirageFuzz dominates the baseline fuzzers under most of the execution time. Such results altogether indicate that MirageFuzz is a rather powerful coverage-guided fuzzer.

We also investigate the effectiveness of exploring unique edges (i.e., edges that can only be explored by a given fuzzer) for all our studied fuzzers. In our evaluation, MirageFuzz can achieve the best performance by exploring 4268 unique edges on top of the whole benchmark suite averagely, which outperforms the top-performing baseline Angora by 62.16% (4268 vs. 2632 edges). Due to the page limit, we present the performance details in our GitHub page [1].

5.3.2 RQ2: the effectiveness of different components in MirageFuzz. To further understand the mechanism adopted by MirageFuzz, in this section, we perform in-depth ablation studies to investigate the effectiveness of the dedicated components designed for MirageFuzz, i.e., the phantom fuzzing and the taint-based mutation mechanism.

Effectiveness of the phantom fuzzing. Investigating the effectiveness of the phantom fuzzing for MirageFuzz is essentially equivalent to investigating the effectiveness of using the condition value derived by the phantom fuzzing for mutating the corresponding condition of the given seed for the source fuzzing. Accordingly, we determine to create a technique variant MirageFuzz$\text{taint}$ of MirageFuzz which tracks the byte offset impacting the unexplored condition of a given seed and then applies random mutation on the corresponding byte offset. Meanwhile, we activate another source fuzzing process to replace the original phantom fuzzing process. Table 1 also presents the edge coverage results of MirageFuzz$\text{taint}$. We can observe that MirageFuzz significantly outperforms MirageFuzz$\text{taint}$ by 23.94%. Moreover, we can also find that both Hawc$\text{MAR}$ and QSYM outperform MirageFuzz$\text{taint}$ by 0.02% and 9.27% respectively. Such results suggest that phantom fuzzing is essential in strengthening the effectiveness of MirageFuzz.

Finding 3: The phantom fuzzing is critical for MirageFuzz to augment its edge coverage performance.

Effectiveness of taint-based mutation mechanism. We create a technique variant MirageFuzz$\text{splice}$ which replaces the taint-based mutation mechanism by randomly identifying a byte offset of a given seed in the source fuzzing and splicing the given seed and a randomly selected seed for the phantom fuzzing at the identified byte offset to generate a mutant for the source fuzzing. Table 1 also presents the edge coverage results of MirageFuzz$\text{splice}$ where MirageFuzz outperforms MirageFuzz$\text{splice}$ by 19.62%. Such a result clearly demonstrates that applying the taint-based mutation mechanism can advance the effectiveness of the phantom fuzzing by precisely positioning the byte offset associated with the unexplored condition and providing the condition value to generate a mutant which can be executed to facilitate the source fuzzing.

Finding 1: MirageFuzz is a rather powerful coverage-guided fuzzer which can significantly and consistently outperform the adopted baseline fuzzers.

Finding 2: The mechanisms adopted by MirageFuzz are potentially more effective than applying constraint solver for exploring program states.

Interestingly, while QSYM and Angora are generally more effective than other baseline fuzzers, the fact that MirageFuzz significantly outperforms them on all benchmark programs without applying a constraint solver indicates that its insight which enhances the exploration power of seeds via dual fuzzing only is potentially even more powerful in exploring program states.
### Table 1: Effectiveness of MirageFuzz

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AFL</th>
<th>AFL++</th>
<th>LaIntel</th>
<th>FairFuzz</th>
<th>MOPT</th>
<th>Havoc</th>
<th>Angora</th>
<th>MirageFuzz\textsubscript{init}</th>
<th>MirageFuzz\textsubscript{plus}</th>
<th>MirageFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>readelf</td>
<td>4511</td>
<td>7137</td>
<td>5523</td>
<td>8839</td>
<td>11880</td>
<td>11883</td>
<td>13228</td>
<td>11742</td>
<td>12126</td>
<td>13611</td>
</tr>
<tr>
<td>nm</td>
<td>2637</td>
<td>5770</td>
<td>3295</td>
<td>4091</td>
<td>5134</td>
<td>5924</td>
<td>5924</td>
<td>6548</td>
<td>6548</td>
<td>5568</td>
</tr>
<tr>
<td>objdump</td>
<td>2843</td>
<td>3551</td>
<td>4741</td>
<td>5753</td>
<td>5786</td>
<td>6607</td>
<td>5923</td>
<td>6314</td>
<td>N/A</td>
<td>7174</td>
</tr>
<tr>
<td>objcopy</td>
<td>3280</td>
<td>3146</td>
<td>2856</td>
<td>4080</td>
<td>7456</td>
<td>7528</td>
<td>8167</td>
<td>N/A</td>
<td>9830</td>
<td>8400</td>
</tr>
<tr>
<td>size</td>
<td>2095</td>
<td>2185</td>
<td>2510</td>
<td>3447</td>
<td>3622</td>
<td>3651</td>
<td>4973</td>
<td>4206</td>
<td>5038</td>
<td>4011</td>
</tr>
<tr>
<td>jhead</td>
<td>161</td>
<td>177</td>
<td>199</td>
<td>161</td>
<td>165</td>
<td>169</td>
<td>784</td>
<td>804</td>
<td>176</td>
<td>753</td>
</tr>
<tr>
<td>pcree2</td>
<td>6827</td>
<td>6446</td>
<td>6510</td>
<td>6400</td>
<td>6209</td>
<td>6566</td>
<td>6277</td>
<td>6516</td>
<td>4082</td>
<td>6178</td>
</tr>
<tr>
<td>pngfix</td>
<td>1044</td>
<td>1043</td>
<td>1835</td>
<td>1033</td>
<td>1152</td>
<td>1045</td>
<td>1988</td>
<td>2093</td>
<td>1774</td>
<td>1761</td>
</tr>
<tr>
<td>strip</td>
<td>3022</td>
<td>3906</td>
<td>3104</td>
<td>4225</td>
<td>6970</td>
<td>6927</td>
<td>6725</td>
<td>N/A</td>
<td>9071</td>
<td>7027</td>
</tr>
<tr>
<td>listaction_d</td>
<td>3770</td>
<td>4118</td>
<td>4046</td>
<td>4164</td>
<td>4042</td>
<td>3831</td>
<td>4101</td>
<td>3889</td>
<td>3391</td>
<td>4099</td>
</tr>
<tr>
<td>libxml2</td>
<td>9657</td>
<td>9015</td>
<td>4468</td>
<td>8085</td>
<td>9559</td>
<td>9268</td>
<td>10346</td>
<td>N/A</td>
<td>9513</td>
<td>9019</td>
</tr>
<tr>
<td>libpng</td>
<td>863</td>
<td>861</td>
<td>2057</td>
<td>1060</td>
<td>1061</td>
<td>868</td>
<td>2021</td>
<td>N/A</td>
<td>1872</td>
<td>1873</td>
</tr>
<tr>
<td>re2</td>
<td>6911</td>
<td>6915</td>
<td>6874</td>
<td>6874</td>
<td>6855</td>
<td>6844</td>
<td>6837</td>
<td>N/A</td>
<td>6143</td>
<td>6460</td>
</tr>
<tr>
<td>swftocxx</td>
<td>4519</td>
<td>4544</td>
<td>4369</td>
<td>4378</td>
<td>4327</td>
<td>4413</td>
<td>4483</td>
<td>4545</td>
<td>4474</td>
<td>4359</td>
</tr>
<tr>
<td>jsoncpp</td>
<td>1010</td>
<td>1010</td>
<td>1002</td>
<td>1012</td>
<td>1010</td>
<td>1010</td>
<td>946</td>
<td>N/A</td>
<td>886</td>
<td>987</td>
</tr>
<tr>
<td>lcms2</td>
<td>888</td>
<td>1395</td>
<td>933</td>
<td>864</td>
<td>864</td>
<td>862</td>
<td>9651</td>
<td>N/A</td>
<td>1083</td>
<td>879</td>
</tr>
<tr>
<td>file</td>
<td>1268</td>
<td>1452</td>
<td>852</td>
<td>1414</td>
<td>1551</td>
<td>1480</td>
<td>1857</td>
<td>1861</td>
<td>1627</td>
<td>1493</td>
</tr>
<tr>
<td>libjpeg-turbo</td>
<td>4213</td>
<td>4193</td>
<td>4573</td>
<td>5281</td>
<td>4180</td>
<td>4623</td>
<td>4745</td>
<td>N/A</td>
<td>3979</td>
<td>4380</td>
</tr>
<tr>
<td>average</td>
<td>3261</td>
<td>3587</td>
<td>3244</td>
<td>3846</td>
<td>4526</td>
<td>4659</td>
<td>5090</td>
<td>4613</td>
<td>4626</td>
<td>4658</td>
</tr>
<tr>
<td>p-value</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>-</td>
</tr>
</tbody>
</table>

Finding 4: The taint-based mutation mechanism is essential for MirageFuzz in facilitating its fuzzing efficacy.

We further investigate the taint-analysis time in our fuzzing campaign (presented in our GitHub page [1] due to the page limit), where it ranges from 1745 to 33589 seconds averagely during 24-hour runs. Notably, even though it costs 33589 seconds for taint analysis on project strip, MirageFuzz still achieves the best edge coverage (i.e., covering 9417 edges) averagely in 24-hour run.

### 5.4 Bug Report and Analysis

In this paper, we obtain all the crashes and then manually identify the buggy location through stack tracing and analyze their respective causes. Accordingly, we derive unique bugs via debugging. We then report our exposed bugs to the developers with the essential information that can help them generate a patch. Overall, applying MirageFuzz exposes 29 previously unknown bugs upon our benchmark suite where 7 were confirmed and 6 were fixed by the corresponding developers. Meanwhile, AFL, AFL++ and MEUZZ detect 2 out-of-memory bugs in project swftocxx, and AFL++, FairFuzz and QSYM expose 2 heap-buffer-overflow bugs in...
project listaction_d. Note that MirageFuzz can expose all the bugs exposed by all other fuzzers. We illustrate all our bug types, e.g., a use-of-uninitialized-value bug refers to using a variable without initialization, in our GitHub pages [1]. Table 2 presents the details of the previous unknown bugs exposed by MirageFuzz.

### Table 2: The bug information

<table>
<thead>
<tr>
<th>Program</th>
<th>Bug Type</th>
<th>Number</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>pcre2</td>
<td>Infinite loop</td>
<td>1</td>
<td>confirmed and fixed</td>
</tr>
<tr>
<td>nm</td>
<td>Infinite loop</td>
<td>1</td>
<td>reported</td>
</tr>
<tr>
<td>jhead</td>
<td>Use-of-uninitialized-value</td>
<td>3</td>
<td>confirmed and fixed</td>
</tr>
<tr>
<td>strip</td>
<td>Out-of-memory</td>
<td>1</td>
<td>confirmed</td>
</tr>
<tr>
<td>pngfix</td>
<td>Use-of-uninitialized-value</td>
<td>1</td>
<td>reported</td>
</tr>
<tr>
<td>listaction_d</td>
<td>Segmentation fault</td>
<td>6</td>
<td>reported</td>
</tr>
<tr>
<td>swftocxx</td>
<td>Segmentation fault</td>
<td>5</td>
<td>reported</td>
</tr>
<tr>
<td></td>
<td>Heap-buffer-overflow</td>
<td>3</td>
<td>reported</td>
</tr>
<tr>
<td></td>
<td>Allocation-size-too-big</td>
<td>1</td>
<td>reported</td>
</tr>
<tr>
<td></td>
<td>Out-of-memory</td>
<td>2</td>
<td>reported</td>
</tr>
</tbody>
</table>

5.4.1 Infinite loop in pcre2test. We have reported a bug on project pcre2 [10]—a set of C functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5 [12]. It was assigned with an issue ID 141 [11] and has been confirmed and fixed by developers. This bug was exposed by running pcre2test, one of the executable programs in project pcre2 with the specified input files only generated by MirageFuzz.

While processing the input files, an infinite loop in function pcre2test.c:process_data(void) occurred as shown in Figure 5.

```
1  int process_data(void)
2  {
3      // ...
4      // p is a section from input file, li is s64, i is s32, needlen and dbuffer_size are u64.
5      li = strtol((const char *)p, &endptr, 10);
6      if (S32OVERFLOW(li)) { return OK; }
7      i = (int32_t)li;
8      if (i-- == 0) { return OK; }
9      // ...
10     replen = CAST8VAR(q) - start_rep;
11     needlen += replen * i;
12     dbuffer_size *= 2;
13     if (needlen >= dbf_size)
14         return OK;
15     // ...
16     while (needlen >= dbf_size)
17         dbuffer_size *= 2;
18     // ...
19     // ...
20   }
21 }
```

Figure 5: Infinite loop in pcre2test.

For the while condition needlen >= dbf_size and the loop body dbf_size *= 2, we analyze that the value of needlen potentially incurs infinite looping due to a possible integer overflow. In fact, one of our input files sets i = -10, which in turn assigns needlen with the value resulting in an infinite loop.

Correspondingly, the developers made a simple fix, i.e., patching i- == 0 as i- <= 0. They commented on this bug as follows:

“A negative repeat value in a pcre2test subject line was not being diagnosed, leading to infinite looping.”

5.4.2 Use-of-uninitialized-value in pngfix. We reported a use-of-uninitialized-value bug in project libpng [8] only exposed by MirageFuzz under the instrumentation by MemorySanitizer [67]. In particular, the bug was exposed by running the generated seed from pngfix, one of the executable programs in project libpng, confirmed with the GitHub issue ID 424 [13] and fixed later.

The buggy code snippet is presented in Figure 6 where the uninitialized value reported by MemorySanitizer comes from png_ptr->big_row_buf and png_ptr->big_prev_row.

```
1  void png_read_start_row(png_structrp png_ptr)
2  {
3      // ...
4      if (png_ptr->interlaced != 0)
5          png_ptr->big_row_buf = (png_bytep)
6          png_malloc(png_ptr,png_ptr->big_row_bytes+48);
7      else
8          png_ptr->big_row_buf = (png_bytep)
9          png_malloc(png_ptr,png_ptr->big_row_bytes+48);
10     png_ptr->big_prev_row = (png_bytep)
11     png_malloc(png_ptr,png_ptr->big_prev_bytes+48);
12     // ...
13 }
```

Figure 6: Use-of-uninitialized-value in pngfix.

The developers believed that this problem was caused by lacking the memory initialization before using the memory requested by malloc and then fixed the bug by invoking memset in the end of the code snippet in Figure 6 with the following feedback:

“In my opinion it is due to the fact that png_malloc just calls malloc but doesn’t initialize the memory. I can work on that and improve it. It would really help to avoid similar issues in the future.”

5.4.3 Use-of-uninitialized-value in jhead. We reported multiple use-of-uninitialized-value bugs of project jhead. These bugs, reported in a GitHub issue (ID 53) [6], were confirmed and fixed.

The relevant buggy code snippet in function ReadJpegSections is shown in Figure 7, where Data is a pointer to an allocated heap memory segment by invoking malloc. However, such a memory segment is not initialized before Data is used in the subsequent procedure, and thus leads to a vulnerability.

```
1  int ReadJpegSections (FILE * infile, ReadMode_t ReadMode)
2  {
3      // ...
4      uchar * Data;
5      // ...
6      Data = (uchar *)malloc(itemlen+20);
7      if (Data == NULL){
8          ErrFatal("Could not allocate memory");
9      }
10     Sections[SectionsRead].Data = Data;
11     // ...
12 }
```

Figure 7: Use-of-uninitialized-value in jhead.

Eventually, the developer generated a patch by invoking memset to initialize the value of the related memory after it is allocated.
5.4.4 Out-of-memory in strip. We have reported one out-of-memory bug as a bugzilla issue with ID 29495 [14] when executing project strip, which was confirmed and fixed by the associated developers.

The function exif.c:rewriteElf_program_header in Figure 8 reveals the relevant buggy code snippet. By using the input generated by our approach, the execution on strip keeps consuming memory and causes an out-of-memory bug. In our evaluation, strip consumes 64 GiB memory in our server in about two minutes.

Similar to malloc, we found that bfd_zalloc is a function that allocates memory in the heap, located in the loop in line 18. The loop only terminates by updating isec surrounded by a conditional code region (lines 8 to 12). Therefore, an out-of-memory bug is triggered if strip fails to enter such code region, i.e., the condition of such a code region cannot be satisfied.

```c
static bool rewrite_elf_program_header
(bfd *ibfd, bfd *obfd, bfd_vma maxpagesize)
{
  // ...
  isec = 0;
  do {
    // ...
    if (IS_CONTAINED_BY_LMA(output_section, segment, map->p_paddr, opb)
      || IS_COREFILE_NOTE(segment, section))
    {
      ++isec;
    }
    // ...
    if (isec < section_count) {
      // bfd_zalloc allocates memory.
      map = (struct elf_segment_map *) bfd_zalloc(obfd, amt);
      // ...
    }
    continue;
  } while (isec < section_count);
  // ...
}
```

Figure 8: Out-of-memory in strip.

The developers fixed this bug by refactoring the whole function to avoid memory overflow. They also commented the bug as follows:

"It’s important that the later tests not be more restrictive. If they are it can lead to the situation triggered by the testcases, where a section seemingly didn’t fit and thus needed a new mapping. It didn’t fit the new mapping either, and this repeated until memory exhausted."

6 THREATS TO VALIDITY

Threats to internal validity. The threat to internal validity lies in the implementation of our approach. To reduce this threat, we reused the source code of the original AFL [78] to construct our basic fuzzing framework when implementing MirageFuzz. Meanwhile, to implement the taint-based mutation mechanism, we also reuse the existing libraries for taint analysis. Moreover, the first three authors manually reviewed MirageFuzz code carefully to ensure its correctness and consistency.

Threats to external validity. The threat to external validity mainly lies in the benchmarks and the baselines used. To reduce this threat, we adopt 18 projects widely used for the evaluations in many popular fuzzers published recently [23, 23, 24, 47, 62, 72]. Furthermore, we also select nine popular baseline fuzzers, including six traditional coverage-guided fuzzers [7, 28, 42, 47, 72, 78] and three constraint-solving-based fuzzers [23, 25, 77] to evaluate the effectiveness of MirageFuzz.

7 RELATED WORK

7.1 Fuzzing

Among all the coverage-guided fuzzers [15, 35, 62, 72–74], AFL [78] is a widely-used baseline by retaining the mutants which can be executed to increase code coverage as seeds for further iterative executions. Many fuzzers are implemented upon AFL. Li et al. [43] proposed Steelix to explore new coverage efficiently by observing more runtime states. Lemieux et al. [42] introduced the concept of rare branches and facilitated the fuzzing efficacy by focusing on rare branches. In order to improve the fuzzing effectiveness, researchers also attempt to integrate dynamic analysis techniques such as taint analysis with fuzzing, e.g., AFL++ [28]. Rawat et al. [58] proposed VUzzer to identify the input format of the target program via taint analysis, for avoiding early termination in fuzzing. Liang et al. [45] proposed PATA, a more advanced taint analysis technique that can identify the loop variables efficiently during fuzzing. Du et al. [27] proposed WindRanger, which leverages the power of deviation basic blocks to facilitate directed grey-box fuzzing. Furthermore, many researchers also propose seed scheduling techniques for improving fuzzing effectiveness. Böhme et al. [18] proposed AFLFast to schedule seeds during fuzzing via a Markov chain model to improve the performance of AFL. She et al. [63] introduced K-scheduler, which schedules seeds according to the reachable edges and potential coverage gain. Zhang et al. [80] utilized path constraint as the guidance function to schedule the seeds for harvesting new edges. Zhang et al. [79] proposed MobFuzz, which models fuzzing as a multi-objective problem via a multi-armed bandit and then schedules the seeds based on a particular optimization goal derived from the chosen objective combination. Meanwhile, Chen et al. [25] proposed MEUZZ to schedule the seeds in hybrid fuzzing based on the knowledge learned from past seed scheduling decisions made on the same or similar programs. Researchers also adopt constraint solvers to explore deep program states. Cadar et al. [19] proposed the fundamental symbolic execution engine Klee for aiding the fuzzers in solving the program constraints during fuzzing via symbolic execution. Accordingly, Yun et al. [77] introduced QSYM to combine a concolic executor for solving complicated program constraints in a selected coverage-guided fuzzer.
to leverage the power of symbolic execution in fuzzing. Kukucka et al. [39] proposed CONFETTI to combine taint analysis and concolic execution to fuzz Java programs. To solve the constraints more efficiently, Chen et al. [22] proposed JIGSAW to evaluate the generated seeds with constraints on a native function produced by Just-in-time compilation. Instead of adopting the SMT-solver as other constraint-solving-based fuzzers, Chen et al. [23] proposed Angora to solve program constraints by a gradient descent algorithm. In addition, Fuzzing is utilized to detect vulnerabilities in specific domains. Shen et al. [64] proposed Drifuzz to fuzz WiFi and Ethernet drivers with concolic executor. Garbelini et al. [30] proposed Brak’Tooth to fuzz arbitrary Bluetooth Classic (BT) devices via constructing a protocol state machine. Shou et al. [65] proposed Corfuzz to fuzz the security policies of browsers by tracking the runtime behaviors of the browsers. Gao et al. [29] incorporated code representation learning and clustering to guide the process of program-synthesis-based JVM fuzzing (such as JavaTailor [82]).

Many existing fuzzers [17, 61, 75, 81, 83] focus on scheduling promising seeds, adopting dynamic analysis techniques or utilizing an additional constraint solver to enhance code coverage. In this paper, we propose MirageFuzz to enhance the exploration capacity of each seed by reducing the program dependencies for conditional statements to reduce the difficulties of accessing their program states.

7.2 Program Transformation

Researchers adopt program transformation for multiple purposes. Bacon et al. [16] proposed multiple ways to optimize programs via transformation in the compiler. Wu et al. [76] proposed AuCS, which utilized the power of program transformation to fix synchronization issues for CUDA programs. Korel et al. [38] utilized program transformation to find program inputs on which a selected element, e.g., a target statement, is executed. Harman et al. [32] generated new tests to improve the performance of search-based testing techniques via program transformation. Chen et al. [21] adopted semantics-preserving program transformation to facilitate the efficacy of symbolic execution. Program transformation is also a common practice for fuzzing. Peng et al. [56] proposed T-Fuzz, which combines symbolic execution and program transformation to explore deep execution paths of the target program. Liu et al. [46] proposed InstruGuard, which detects and fixes the errors generated by transforming the target program for obtaining coverage information via static analysis on target binaries and rewriting transformation rules. Wang et al. [69] introduced RIFF to reduce the fuzzing overhead generated by program coverage measurement transformation via static program analysis. Menendez et al. [49] proposed HashFuzz which utilizes hash functions for semantics-preserving program transformation to target programs for generating more diverse inputs. Dinesh et al. [26] proposed RetroWrite, which utilizes static analysis to transform target programs to reduce the performance overhead incurred by sanitizers in fuzzing. Nagy et al. [52] introduced a new program transformation rule to eliminate the needless coverage tracing for coverage-guided fuzzers. Hsu et al. [34] proposed a lightweight program transformation strategy to reduce the fuzzing overhead incurred by tracing the coverage information.

Mutation testing [54, 68] is a type of software testing in which certain statements of the source code are changed/mutated to check if the test cases are able to find errors in the source code. In mutation testing, test cases are expected to reject mutant (i.e., mutated program) by causing the behavior of the original program to differ from the mutant. Specifically, Papadakis et al. [55] and Chekam et al. [20] have studied the fault revelation ability of mutation testing and found that the higher mutation scores are, the stronger the fault revelation ability of mutation testing is. While mutation testing is typically adopted for evaluating the quality of test suites, the adopted program transformation from mutation testing enlightens researchers on facilitating the fuzzing efficacy. Groce et al. [31] has shown that fuzzing the mutants of the target program can allow a fuzzer to explore more behaviors than spending the entire fuzzing budget on the original target. Qian et al. [57] utilized mutation scores as additional feedback to guide fuzzing for bug detection.

While many fuzzers adopt program transformation for reducing runtime overhead, we leverage the power of program transformation to create a phantom program for enhancing the exploration capacity of all seeds.

8 CONCLUSION

In this paper, we propose the concept of phantom program, which is built to mitigate the over-compliance of program dependencies to enhance the exploration capacity of all seeds. Accordingly, we build a coverage-guided fuzzer namely MirageFuzz which performs dual fuzzing for the original program and the phantom program simultaneously and adopts the taint-based mutation mechanism to generate new mutants by combining the resulting seeds from dual fuzzing via taint analysis. To evaluate the effectiveness of MirageFuzz, we select 18 frequently used projects to form our benchmark suite and nine popular open source fuzzers to form our baseline fuzzers. The evaluation results show that MirageFuzz outperforms the baseline fuzzers from 13.42% to 77.96% in terms of edge coverage averagely in our benchmark. MirageFuzz also exposes 29 previously unknown unique bugs where 7 of them have been confirmed and 6 have been fixed by the corresponding developers.

9 DATA AVAILABILITY

The source code of the MirageFuzz implementation is available in our GitHub page [4]. All evaluation details and bug reports are also presented in the GitHub page [1].

ACKNOWLEDGEMENT

This work is partially supported by Guangdong Provincial Key Laboratory (Grant No. 2020B121201001) and National Natural Science Foundation of China Grant Nos. 62002256, 62232001. This work is also partially supported by Kuaishou.
Enhancing Coverage-Guided Fuzzing via Phantom Program

ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA


[57] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, and Lihua Guo. 2022. Inves-...