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

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

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

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:
Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. 2023. Enhancing Coverage-Guided Fuzzing via Phantom Program. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616294

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

*Mingyuan Wu is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China.
†Yuqun Zhang is the corresponding author. He is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, China

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

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616294

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 `user` 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[0]` 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 4 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

```
1  void Origin(char *user){
2    user[4] = '\0';
3    if (user[0] == 'M') {
4      if (user[1] == 'A')
5        if (user[2] == 'Z')
6          if (user[3] == 'E')
7          // crash
8    } else {
9      ...
10     // other code
11   }
12   return;
13 }
14
```

(a) The original code

```
1    void Phantom(char *user){
2      user[4] = '\0';
3      if (user[0] == 'M');
4      if (user[1] == 'A');
5      if (user[2] == 'Z');
6      if (user[3] == 'E')
7         // crash
8      if (user[0] != 'M'){
9       ...
10      // other code
11     }
12     return;
13   }
14
```

(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 line 6 in Figure 1a, i.e., it satisfies line 5 but fails to satisfy line 6. We can attempt to obtain the byte offset of the running seed corresponding to line 6, i.e., the branch condition "`user[3] == 'E'`", via taint analysis. If we could also identify a seed which can be executed to explore the same branch condition in Figure 1b, we could then apply taint analysis again on that seed to obtain the operand value and use it to update the byte offset of the seed running in Figure 1a. At last, the resulting mutant can be executed to satisfy line 6 in Figure 1a and thus trigger the crash, indicating that the power of exploring program states of the original seed is improved. Accordingly, in this paper, we are inspired to propose a technique which aims at reducing program dependencies for enhancing the exploration power of seeds on program states.

Note that our mission in Figure 1 is seemingly close to advance 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 or integrating constraint solver [51]. 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 ❶ 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 (❷, Section 3.2). Specifically, during iterative executions of the *source fuzzing* under a given seed S, *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* (❸, 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", &PO)` of block ⑤ in entry block ① after instruction relocation, the value of variable `ret` calculated via instruction `sscanf(argv[++n], "%d", &FQ)` of block ② is changed since its variable `n` has already been updated in entry block ①. Accordingly, the semantics of condition `bool cmp2 = (ret != 1)` for block ③ 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 `blocks`) with the conditional instruction of each basic block (represented as `branchCon`, lines 2 to 4). Next, for each `block`,

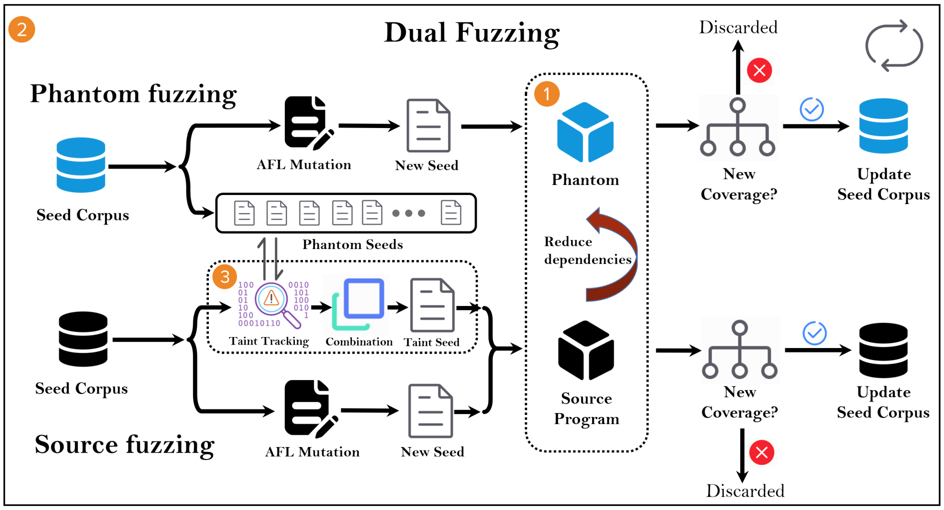

Figure 2: The workflow of *MirageFuzz*

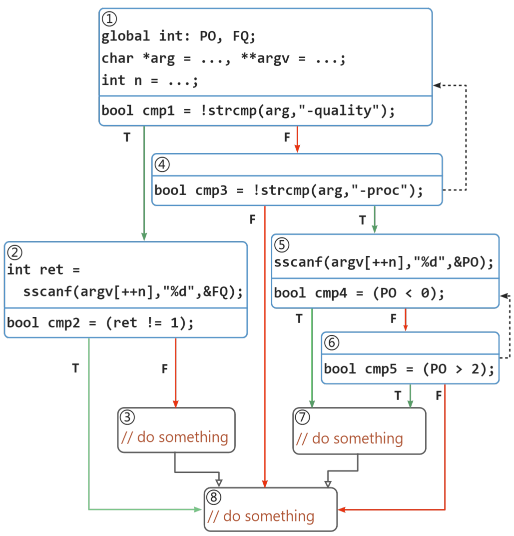

Figure 3: A simplified real-world example from *jhead*

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

Algorithm 1 Dependency Reduction Mechanism

Input: *source*
Output: *phantom*

1: **function** ReduceDependencies
2:     *blocks* ⟵ *getBasicBlocks*(*source*)
3:     **for** *each block* **in** *blocks* **do**
4:         *branchCon* ⟵ *getBranchExpression*(*block*)
5:         *collectedBlocks* ⟵ *slicingBasicBlocks*(*branchCon*)
6:         *remainBlocks* ⟵ $\{b \in$ *collectedBlocks* ∥ *violateSemanticsAfterRelocating*(*b*) *is* **True** ∧ *isDominator(b, branchCon) is **True***$\}$
7:         **if** *remainBlocks* **is** $\emptyset$ **then**
8:             *entrance* ⟵ *getEntryBlock*(*blocks*)
9:         **else**
10:             *entrance* ⟵ *randomChoice(remainBlocks)*
11:             **for** *each B* **in** *remainBlocks* **do**
12:                 *entrance* ⟵ *entrance dominates B ? B : entrance*
13:         *instructions* ⟵ *getRelatedInstructions*(*collectedBlocks*)
14:         *move instructions to entrance*
15:     *phantom* ⟵ *reconstruct*(*blocks*)
16:     **return** *phantom*

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

Algorithm 2 Dual Fuzzing

```
Input: initialSeed, budget
Output: None
1: function FuzzingSourceProgram
2:     seeds ← {initialSeed}
3:     while fuzzing time not exceed budget do
4:         for each seed in seeds do
5:             mutant ← AFLMutation(seed)
6:             if mutant has new edges then
7:                 seeds ← seeds ∪ {mutant}
8:             edges ← getUnexploredEdges(seed)
9:             phantomSeeds ← requestSeedsFromPhantom(edges)
10:            taintSeeds ← GenerateSeedWithTaint(seed, phantomSeeds)
11:            for each tSeed in taintSeeds do
12:                if tSeed has new edges then
13:                    seeds ← seeds ∪ {tSeed}
14:    return None
15: function FuzzingPhantom
16:    seeds ← {initialSeed}
17:    edgeDic ← [[ edge ⇒ {}]]
18:    while fuzzing time not exceed budget do
19:        if has requests from source fuzzing then
20:            resp ← {}
21:            for each edge in requests do
22:                resp ← resp ∪ edgeDic[[edge]]
23:            Send resp to source fuzzing
24:        for each seed in seeds do
25:            mutant ← AFLMutation(seed)
26:            if mutant has new edges then
27:                seeds ← seeds ∪ {mutant}
28:                for each edge in mutant do
29:                    edgeDic[[edge]] ← edgeDic[[edge]] ∪ {mutant}
30:    return None
```

executed to explore such collected branches in the *phantom fuzzing* for later generating new seeds to advance the *source fuzzing*.

Algorithm 2 presents the details for the dual fuzzing. In particular, for the *source fuzzing*, we first adopt the AFL mutation strategy [72] to generate mutants out of our initial seed corpus (lines 2 to 5). If any mutant is executed to explore new edges, it is added into the seed corpus for further iterative executions (lines 6 to 7). Meanwhile, we derive the unexplored program edges adjoining the explored program states by executing the given mutant, and check whether they can be explored by executing any of the seeds generated from the *phantom fuzzing* (lines 8 to 9). If such a seed exists, we then adopt the taint-based mutation mechanism (illustrated later) to generate a new seed for the future *source fuzzing* (lines 10 to 13),

For the *phantom fuzzing*, we create a dictionary `edgeDic` to store the information of the explored edges with their corresponding executed seeds (lines 16 to 17). For each iterative execution, we first check the real-time unexplored edges from the *source fuzzing* (lines 18 to 19). Then, we iterate each unexplored edge to find whether it has been already explored by executing any seed generated by the *phantom fuzzing* (lines 20 to 23). Similar to the *source fuzzing*, we also adopt the AFL mutation strategy to generate mutants (lines 24 to 25). If executing any mutant explores new edges, it is added to the seed corpus where `edgeDic` is updated (lines 28 to 29).

Algorithm 3 Taint-based Mutation Mechanism

```
Input: sourceSeed, phantomSeeds
Output: taintSeeds
1: function GenerateSeedWithTaint
2:     taintSeeds ← {}
3:     exploredEdges ← getAllExploredEdgesFromSource()
4:     for each mSeed in phantomSeeds do
5:         for each edge in mSeed's execution path do
6:             if edge not in exploredEdges then
7:                 taintPos ← taintPosition(edge, sourceSeed)
8:                 value ← taintContent(edge, mSeed)
9:                 mutant ← sourceSeed
10:                mutant[taintPos] ← value
11:                taintSeeds ← taintSeeds ∪ {mutant}
12:    return taintSeeds
```

In Figure 3, assume a `seed` is executed to explore the path [①,④,⑤,⑦,⑧] for the *source fuzzing*. Then we can derive the unexplored edges adjoining the explored path, i.e., ①→②, ④→⑧, and ⑤→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.

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* with its value derived by the *phantom 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 `taintPos` corresponding to the conditional instruction by taint analysis in the `sourceSeed` (lines 6 to 7). Subsequently, we also obtain the corresponding condition `value` by taint analysis in the given `mSeed` from `phantomSeeds` (line 8). To illustrate, note that to prevent the misallignment between the byte offset of `mSeed` corresponding to the condition `value` and `taintPos`, we activate two taint analysis processes for `sourceSeed` and `mSeed` respectively. Specifically, we first apply taint analysis in the *source fuzzing* for obtaining the byte offset of the seed (i.e., `sourceSeed`) corresponding to the given unexplored branch condition from the source program. If we could identify a seed (i.e., `mSeed`) which can be executed to explore such a branch condition 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 `P0` that determines the transition ⑤→⑥ or ⑤→⑦. We further figure out that the value of `P0` in S' is 14 via taint analysis on S'. Eventually, we replace the value of `P0` in S with 14 to generate a new seed for exploring the new edge ⑤→⑥ 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 *libdft* [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 branch instruction to monitor whether any of its associated edges has been explored by observing the associated sink state of such branch. If the sink of the given branch instruction is not reached, it indicates that the corresponding edge is unexplored. Therefore, combining with the real-time collected explored edges, we can derive the unexplored edges adjoining them.

4.2 Dynamic Taint Analysis

We adopt *libdft* [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 *libdft* implemented the taint propagation rules for 146 instructions, their default taint propagation rules still cannot cover our required instructions, e.g., `bswap` (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 *libdft* 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` [66] 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], $Havoc_{MAB}$ [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 LafIntel (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].

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

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.

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

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 $MirageFuzz_{taint}$ 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 $MirageFuzz_{taint}$. We can observe that *MirageFuzz* significantly outperforms $MirageFuzz_{taint}$ by 23.94%. Moreover, we can also find that both $Havoc_{MAB}$ and QSYM outperform $MirageFuzz_{taint}$ 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 $MirageFuzz_{splice}$ 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 $MirageFuzz_{splice}$ where *MirageFuzz* outperforms $MirageFuzz_{splice}$ 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*.

Table 1: Effectiveness of *MirageFuzz*

Benchmark	AFL	AFL++	LafIntel	FairFuzz	MOPT	$Havoc_{MAB}$	QSYM	MEUZZ	Angora	$MirageFuzz_{taint}$	$MirageFuzz_{splice}$	*MirageFuzz*
readelf	4511	7137	5523	8839	11537	12841	11880	11883	13228	11742	12126	**13611**
nm	2657	3770	3205	4091	5134	5924	6548	5724	4471	5528	5389	**7584**
objdump	2843	3551	3471	5753	5780	6007	6314	N/A	5265	5923	6174	**7735**
objcopy	3240	3146	2856	4080	7456	7528	8167	N/A	9830	7379	8400	**10473**
size	2095	2185	2510	3447	3622	3651	4973	4206	5038	4011	3999	**5469**
jhead	161	177	199	161	165	169	784	804	176	753	767	**858**
pcre2	6027	6446	6510	6460	6209	6566	6277	6516	4082	6178	6555	**6714**
pngfix	1044	1043	1835	1033	1152	1045	1988	2093	1774	1761	1778	**2195**
strip	3022	3606	3104	4225	6970	6927	6725	N/A	9071	7027	7357	**9417**
listaction_d	3770	4118	4046	4164	4042	3831	4101	3889	3391	4097	3914	**4194**
libxml2	9657	9015	4468	8085	9559	9268	10346	N/A	7053	9019	8991	**10992**
libpng	863	861	2057	1060	1061	868	2021	N/A	1872	1873	2199	**2313**
re2	6911	**6915**	6874	6874	6855	6844	6837	N/A	6143	6460	6877	6892
swftocxx	4519	4544	4369	4378	4327	4413	4483	4545	4474	4359	4551	**4675**
jsoncpp	1010	1010	1002	1012	1010	1010	946	N/A	886	987	1001	**1023**
lcms2	888	1395	935	864	864	862	2631	N/A	1083	879	1035	**2824**
file	1268	1452	852	1414	1551	1480	1857	1861	1627	1493	1396	**1970**
libjpeg-turbo	4213	4193	4573	3281	4180	4623	4745	N/A	3797	4380	4658	**4973**
average	3261	3587	3244	3846	4526	4659	5090	4613	4626	4658	4826	**5773**
p-value	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	-

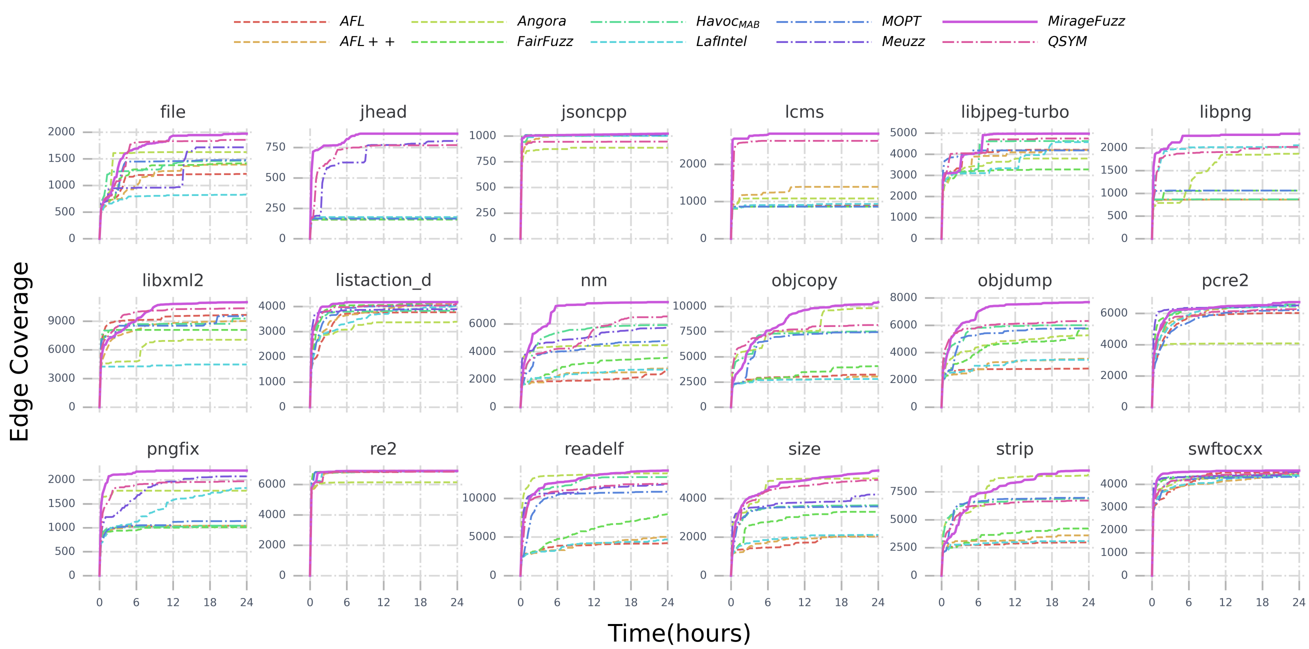

Figure 4: The edge exploration trends of all fuzzers

> *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 respectively 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

Program	Bug Type	Number	Status
pcre2	Infinite loop	1	*confirmed and fixed*
nm	Infinite loop	1	*reported*
jhead	Use-of-uninitialized-value	3	*confirmed and fixed*
strip	Out-of-memory	1	*confirmed and fixed*
pngfix	Use-of-uninitialized-value	1	*confirmed and fixed*
	Infinite loop	1	*confirmed*
listaction_d	Segmentation fault	6	*reported*
	Heap-buffer-overflow	3	*reported*
swftocxx	Segmentation fault	5	*reported*
	Heap-buffer-overflow	4	*reported*
	Allocation-size-too-big	1	*reported*
	Out-of-memory	2	*reported*

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.

```
int process_data(void)
{
  // ...
  // p is a section from input file, li is s64, i is s32, needlen and
     dbuffer_size are u64.
  li = strtol((const char *)p, &endptr, 10);
  if (S32OVERFLOW(li)) { return OK; }
  i = (int32_t)li;
  if (i-- == 0) { return OK; }
  // ...
  replen = CAST8VAR(q) - start_rep;
  needlen += replen * i;

  if (needlen >= dbuffer_size)
  {
   // ...
   while (needlen >= dbuffer_size)
     dbuffer_size *= 2;
   // ...
  }
  // ...
 }
```

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 MemorySantizer [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 Memory sanitizer comes from `png_ptr->big_row_buf` and `png_ptr->big_prev_row`.

```
void png_read_start_row(png_structrp png_ptr)
{
   // ...
   if (png_ptr->interlaced != 0)
     png_ptr->big_row_buf = (png_bytep)
      png_calloc(png_ptr,row_bytes+48);
   else
     png_ptr->big_row_buf = (png_bytep)
      png_malloc(png_ptr,row_bytes+48);
   png_ptr->big_prev_row = (png_bytep)
     png_malloc(png_ptr,row_bytes+48);
   // ...
}
```

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.

```
int ReadJpegSections (FILE * infile, ReadMode_t ReadMode)
 {
  // ...
  uchar * Data;
  // ...
  Data = (uchar *)malloc(itemlen+20);
  if (Data == NULL){
    ErrFatal("Could not allocate memory");
  }
  Sections[SectionsRead].Data = Data;
  // ...
 }
```

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.

> *"Or at least that should fix it. ..., but I could see how this could be triggered."*

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:rewrite_elf_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.

```
1  static bool rewrite_elf_program_header
2    (bfd *ibfd, bfd *obfd, bfd_vma maxpagesize)
3  {
4    // ...
5    isec = 0;
6    do {
7      // ...
8      if (IS_CONTAINED_BY_LMA(output_section, segment, map->p_paddr, opb)
        ||
9        IS_COREFILE_NOTE(segment, section)) {
10       // ...
11       ++isec;
12     }
13
14     // ...
15     if (isec < section_count) {
16       // ...
17       // bfd_zalloc allocates memory.
18       map = (struct elf_segment_map *) bfd_zalloc(obfd, amt);
19       // ...
20     }
21     continue;
22   } while (isec < section_count);
23   // ...
24 }
25
```

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*.

Threats to construct validity. The threat to construct validity mainly lies in the metrics used. To reduce this threat, we determine to follow many prior work [23, 24, 61, 62] by using the edge coverage to represent code coverage. Furthermore, we present the crashes exposed by all studied fuzzers to demonstrate the advantages of *MirageFuzz*. Notably, *MirageFuzz* can incur quite strong performance gain under both metrics compared with other fuzzers.

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 BrakTooth to fuzz arbitrary Bluetooth Classic (BT) devices via constructing a protocol state machine. Shou et al. [65] proposed Corbfuzz 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].

ACKNOWLEGEMENT

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.

REFERENCES

[1] 2022. All experiments detail in the paper. https://github.com/WorldExecute/exprs.

[2] 2022. Control-flow graph generating pass of LLVM. https://llvm.org/docs/Passes.html#dot-cfg-print-cfg-of-function-to-dot-file.

[3] 2022. Dominator and Immediate dominator, WiKipedia. https://en.wikipedia.org/wiki/Dominator_(graph_theory).

[4] 2022. Github Repository. 2022. MirageFuzz. https://github.com/WorldExecute/fuzzer.

[5] 2022. jhead: a simple command line tool for displaying and some manipulation of EXIF header data embedded in Jpeg images from digital cameras. https://github.com/Matthias-Wandel/jhead.

[6] 2022. jhead use-of-uninitialized-value bug issue. https://github.com/Matthias-Wandel/jhead/issues/53.

[7] 2022. laf-intel instrumentation. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.laf-intel.md.

[8] 2022. libpng - library for use in applications that read, create, and manipulate PNG. https://github.com/glennrp/libpng.

[9] 2022. Memory SSA in LLVM. https://llvm.org/docs/MemorySSA.html.

[10] 2022. PCRE2 - Perl-Compatible Regular Expressions. https://github.com/PCRE2Project/pcre2.

[11] 2022. pcre2 infinite loop bug issue. https://github.com/PCRE2Project/pcre2/issues/141.

[12] 2022. Perl - a highly capable, feature-rich programming language. https://www.perl.org/.

[13] 2022. pngfix use-of-uninitialized-value bug issue. https://github.com/glennrp/libpng/issues/424.

[14] 2022. strip out-of-memory bug issue. https://sourceware.org/bugzilla/show_bug.cgi?id=29495.

[15] 2023. LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[16] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (dec 1994), 345–420. https://doi.org/10.1145/197405.197406

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[19] 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.

[20] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, 597–608. https://doi.org/10.1109/ICSE.2017.61

[21] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[22] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*. 18–35. https://doi.org/10.1109/SP46214.2022.9833796

[23] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[24] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 499–513.

[25] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. {MEUZZ}: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 77–92.

[26] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1497–1511. https://doi.org/10.1109/SP40000.2020.00009

[27] Zhengjie Du, Yuekang Li, Yang Liu, Bing Mao, Ligeng Chen, Jian Guo, Zhongling He, Dongliang Mu, C Pang, R Yu, et al. 2022. WindRanger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks. In *2022 IEEE/ACM 44st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.

[28] 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)*.

[29] Tianchang Gao, Junjie Chen, Yingquan Zhao, Yuqun Zhang, and Lingming Zhang. 2023. Vectorizing Program Ingredients for Better JVM Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 526–537.

[30] Matheus E Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 2022. {BrakTooth}: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 1025–1042.

[31] Alex Groce, Goutamkumar Tulajappa Kalburgi, Claire Le Goues, Kush Jain, and Rahul Gopinath. 2022. Registered report: First, fuzz the mutants. In *International Fuzzing Workshop, ser. FUZZING*, Vol. 22.

[32] Mark Harman, André Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. 2008. Testability transformation–program transformation to improve testability. In *Formal methods and testing*. Springer, 320–344.

[33] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/3460319.3464795

[34] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*.

[35] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and Improving Hybrid Fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 410–422. https://doi.org/10.1109/ICSE48619.2023.00045

[36] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 121–132.

[37] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[38] B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. 2005. Data dependence based testability transformation in automated test generation. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*. 10 pp.–254. https://doi.org/10.1109/ISSRE.2005.16

[39] James Kukucka, Lux00ED;s Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 438–450. https://doi.org/10.1145/3510003.3510628

[40] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

[41] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[42] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

[43] 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*. 627–637.

[44] 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.

[45] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1–17. https://doi.org/10.1109/SP46214.2022.9833594

[46] Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. 2021. InstruGuard: Find and Fix Instrumentation Errors for Coverage-based Greybox Fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 568–580. https://doi.org/10.1109/ASE51524.2021.9678671

[47] 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.

[48] Thomas W MacFarland and Jan M Yates. 2016. Mann–whitney u test. In *Introduction to nonparametric statistics for the biological sciences using R*. Springer, 103–132.

[49] Hector D. Menendez and David Clark. 2021. Hashing Fuzzing: Introducing Input Diversity to Improve Crash Detection. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3100858

[50] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.

[51] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[52] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 787–802. https://doi.org/10.1109/SP.2019.00069
[53] Diego Novillo et al. 2007. Memory SSA-a unified approach for sparsely representing memory operations. In *Proceedings of the GCC Developers' Summit*. Citeseer, 97–110.
[54] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.
[55] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering*. 537–548.
[56] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 697–710. https://doi.org/10.1109/SP.2018.00056
[57] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, and Lihua Guo. 2022. Investigating Coverage Guided Fuzzing with Mutation Testing. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware (Internetware '22)*. Association for Computing Machinery, New York, NY, USA, 272–281. https://doi.org/10.1145/3545258.3545285
[58] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/
[59] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.
[60] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26
[61] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. MTFuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 737–749.
[62] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817.
[63] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2194–2211. https://doi.org/10.1109/SP46214.2022.9833761
[64] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. 2022. Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds. In *31st USENIX Security Symposium (USENIX Security 22)*. 1275–1290.
[65] Chaofan Shou, Ismet Burak Kadron, Qi Su, and Tevfik Bultan. 2021. Corbfuzz: Checking browser security policies with fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 215–226.
[66] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. 2018. *Operating System Concepts, 10e Abridged Print Companion*. John Wiley & Sons.
[67] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 46–55.
[68] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
[69] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. {RIFF}: Reduced Instruction Footprint for {Coverage-Guided} Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 147–159.
[70] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.
[71] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.
[72] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*.
[73] 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
[74] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 56–68. https://doi.org/10.1109/ICSE48619.2023.00017
[75] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Yangwei Guo, and Yuqun Zhang. 2023. SJFuzz: Seed & Mutator Scheduling for JVM Fuzzing. In *2023 The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[76] Mingyuan Wu, Lingming Zhang, Cong Liu, Shin Hwei Tan, and Yuqun Zhang. 2019. Automating CUDA synchronization via program transformation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 748–759.
[77] 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.
[78] Michał Zalewski. 2020. American Fuzz Lop. https://github.com/google/AFL.
[79] G Zhang, P Wang, T Yue, X Kong, S Huang, X Zhou, and K Lu. 2022. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.
[80] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. 2022. Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1658–1668. https://doi.org/10.1145/3510003.3510063
[81] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 132–142. https://doi.org/10.1145/3238147.3238187
[82] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1133–1144.
[83] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 347–358.

Received 2023-02-02; accepted 2023-07-27