2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678671

InstruGuard: Find and Fix Instrumentation Errors for Coverage-based Greybox Fuzzing

Yuwei Liu[1,2†], Yanhao Wang[3†], Purui Su[1,2], Yuanping Yu[1,2] and Xiangkun Jia[1,2*]
[1]TCA/SKLCS, Institute of Software, Chinese Academy of Sciences
[2]School of Cyber Security, University of Chinese Academy of Sciences
[3]QiAnXin Technology Research Institute
{yuwei2018, purui, yuanping2017, xiangkun}@iscas.ac.cn wangyanhao@qianxin.com

***Abstract*—As one of the most successful methods at vulnerability discovery, coverage-based greybox fuzzing relies on the lightweight compile-time instrumentation to achieve the fine-grained coverage feedback of the target program. Researchers improve it by optimizing the coverage metrics without questioning the correctness of the instrumentation. However, instrumentation errors, including missed instrumentation locations and redundant instrumentation locations, harm the ability of fuzzers. According to our experiments, it is a common and severe problem in various coverage-based greybox fuzzers and at different compiler optimization levels.**

In this paper, we design and implement InstruGuard, an open-source and pragmatic platform to find and fix instrumentation errors. It detects instrumentation errors by static analysis on target binaries, and fixes them with a general solution based on binary rewriting. To study the impact of instrumentation errors and test our solutions, we built a dataset of 15 real-world programs and selected 6 representative fuzzers as targets. We used InstruGuard to check and repair the instrumented binaries with different fuzzers and different compiler optimization options. To evaluate the effectiveness of the repair, we ran the fuzzers with original instrumented programs and the repaired ones, and compared the fuzzing results from aspects of execution paths, line coverage, and real bug findings. The results showed that InstruGuard had corrected the instrumentation errors of different fuzzers and helped to find more bugs in the dataset. Moreover, we discovered one new zero-day vulnerability missed by other fuzzers with fixed instrumentation but without any changes to the fuzzers.

***Index Terms*—Software Security, Fuzzing, Instrumentation**

I. Introduction

Coverage-based greybox fuzzing has become one of the most popular techniques for software vulnerability discovery due to its ease of use and efficiency [35]. Taking the state-of-the-art fuzzer AFL (American Fuzzy Lop) [52] and its popular family tools [9–11, 13, 14, 17, 18, 23, 24, 27, 36, 44, 47, 50, 53, 54] as examples, the workflow of these greybox fuzzers can roughly be divided into two main stages: instrumentation and fuzzing loop, as shown in Figure 1. The instrumentation codes injected into compiled programs are used to capture branch (edge) coverage or other features while fuzzing. It provides a simple way for fuzzer to automatically mutate input files towards more coverage based on the captured feedback. In this way, coverage-based greybox fuzzing could learn the input format and test the deeper program logic without prior knowledge.

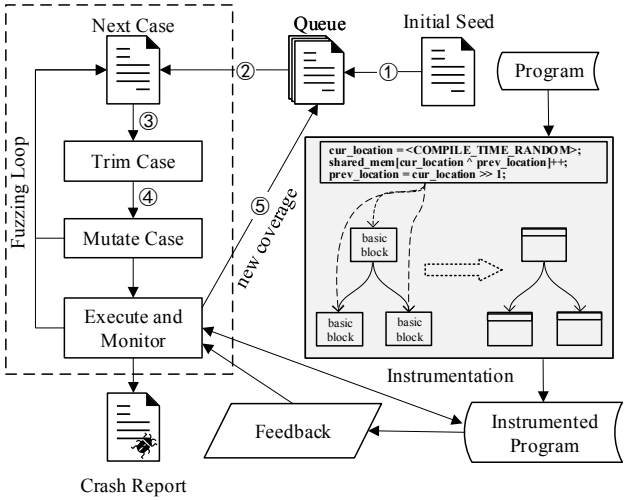


Figure 1: The Workflow of Coverage-based Greybox Fuzzing.

As instrumentation feedback is crucial to greybox fuzzer, researchers propose lots of approaches to improve it. Parts of them enhance the function of the injected code to record more behavior information of the program for detecting specific vulnerabilities, such as concurrency [48] and memory corruption bugs [10, 46, 47]. The other of them [10, 18, 49] improve the sensitivity of the instrumentation feedback for all branches to perceive subtle changes of different program states. All of the current work strongly relies on an assumption that the instrumentation for getting coverage feedback is complete and accurate. However, according to our analysis, the assumption is not always correct.

The current greybox fuzzers provide two methods to inserts instrumentation into each basic block to get accurate edge coverage feedback. One is the assembly-level rewriting approach. The other is compiler-level instrumentation, which leverages the plugin of compilers such as LLVM [22]. However, some factors, such as compiler optimizations, result in basic block merging and other impacts that bring a side effect of missing instrumentation locations and redundant instrumentation locations, and affect the completeness and

† co-leading authors. * corresponding author.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00057

accuracy of the instrumentation feedback for both methods. In the worst-case scenario of missing instrumentation locations of some basic blocks, the fuzzers cannot perceive whether the missing parts are executed and lose the chance to find the potential vulnerabilities in the lost code fragments.

In this work, we try to find and fix errors of instrumentation and set a series of experiments to figure out the impacts on fuzzing. For explicitness of our research, we name the incorrect instrumentation locations in the instrumented binaries as "instrumentation errors", and break the problem into three research questions as follows:

- **RQ1**: *How serious are instrumentation errors?*
- **RQ2**: *Can we fix instrumentation errors of fuzzers?*
- **RQ3**: *Does the fixed instrumentation benefit to fuzzing?*

To answer the research questions, we built a dataset of 15 real-world programs collected from recent fuzzing papers as our targets [10, 12, 13, 18, 27, 29, 36, 46, 50] and selected 6 representative greybox and whitebox fuzzers (i.e., AFL [52], FairFuzz [24], MOPT [27], Memlock [47], AFL++ [16], and Angora [10]) to study the impacts. We instrumented the target programs following the fuzzers' instructions with their instrumentation methods, including assembly-level instrumentation and compiler-level instrumentation, then checked instrumentation errors in the binaries. We also studied the impacts of different compiler optimization options. Concretely, we traversed each basic block in the program based on IDA Pro [2] and compared it with the specific patterns to judge whether it was instrumented or not heuristically. The results showed that the problem is common in the real world as it exists in both types of instrumentation methods of different fuzzers and at all compiler optimization levels.

A straightforward solution to fix instrumentation errors is controlling the compiler options. However, according to our experiments, the compiler-level instrumentation errors could not be eliminated effectively. This paper presents a general method based on binary rewriting and we implement a prototype tool called InstruGuard to find and fix instrumentation errors of coverage-based greybox fuzzers.

For evaluating our solutions, we ran a series of fuzzing experiments on the dataset we built. More specifically, we used code coverage, path number, and real bugs[1] as the metrics to compare the results of original instrumented programs and the fixed ones with different fuzzers. The results of 72 hours $\times$ 5 times running showed that our fix could ensure the correctness of the instrumentation implementation. Although we are not improving coverage-feedback greybox fuzzing directly, our solutions are beneficial for fuzzing. We found one new vulnerability missed by other fuzzers just with the fixed instrumentation but without any changes to the original fuzzer. Overall the paper makes the following contributions:

- We point out that instrumentation errors, including missed instrumentation locations and redundant instrumentation locations, are common for different fuzzers and different compiler optimization options, and impact coverage-based greybox fuzzing seriously.
- We propose a general solution for instrumentation errors based on binary rewriting. Based on the solution, we design and implement an open-source and pragmatic platform to find and fix instrumentation errors.
- We built a dataset of real-world programs and evaluated the effectiveness of InstruGuard by fuzzing the fixed programs and the original instrumented programs without any modification of the fuzzers. The results showed that we had corrected the instrumentation of coverage-based greybox fuzzing and helped to find more vulnerabilities.

To foster future research, we will release the source code of InstruGuard and the dataset at https://github.com/Marsman1996/instruguard.

[1]Real bugs represent the vulnerabilities found by fuzzers and are manually verified. We associate each real bug with the corresponding CVE-ID or bug issue number.

II. Background

A. Coverage-based greybox Fuzzing

Fuzzing was proposed in the 1990s [28] and developed for decades. Among kinds of fuzzers, coverage-based greybox fuzzers (e.g., AFL [52]) attract more attention recently because of their high efficiency and ease of use. Based on a modified form of edge coverage to effortlessly pick up subtle, local-scale changes to program control flow, coverage-based greybox fuzzing could mutate towards more program paths and find more vulnerabilities.

To get the coverage feedback, coverage-based greybox fuzzers implement instrumentation while compiling. The code for collecting coverage information is inserted into the target program by two methods when the source code is available, i.e., assembly-level instrumentation and compiler-level instrumentation. We will use AFL as the representation of coverage-based greybox fuzzers to introduce the details of instrumentation in the following paragraphs.

Assembly-level instrumentation. AFL uses wrappers (i.e., `afl-gcc` and `afl-clang`) for two normal compilers (i.e., `gcc` and `clang`) to conduct assembly-level instrumentation and produce binaries. They parse the assembly file line by line and modify it during the compilation stage according to the following rules. *Rule1*: If the line is a function label, branch destination label, or conditional jump instruction, they will add instrumentation. It is mainly because these labels and instructions mark the boundaries of the basic blocks. *Rule2*: If the line is in the section other than `text`[2] or after `.p2align`[3], they will leave this basic block un-instrumented even though the line satisfies the *Rule1*.

Compiler-level instrumentation. The LLVM mode of AFL leverages `afl-clang-fast` to do compiler-level instrumentation via loading LLVM pass while compiling. It walks

[2]The `text` section is used for keeping the actual assembly code of a program. Hence, AFL only inserts instrumentation into this section.

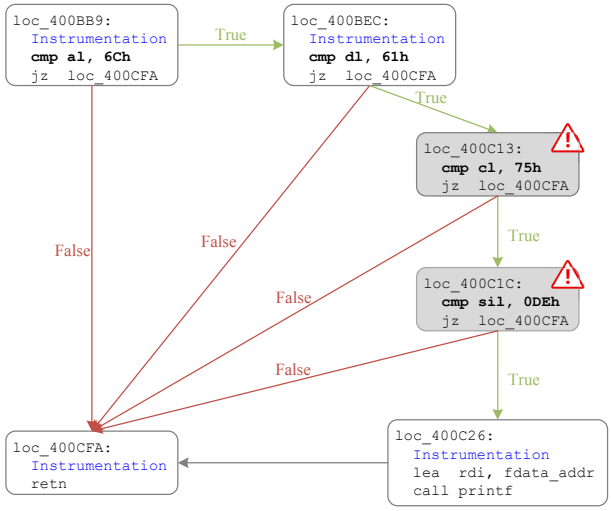
[3]AFL does not intend to instrument the basic blocks after `.p2align` to reduce unnecessary instrumentation while compiling the program under OpenBSD.

```
1 fread(hdr, sizeof(file_header), 1, f);
2 if (hdr->magic != MAGIC) exit(1);
3 entry *ent = (entry *) malloc(sizeof(entry));
4 fread(ent, sizeof(entry), 1, f);
5 if (ent->type == TYPEA) {
6   if (ent[0] == 0x6c) {
7     if(ent[1] == 0x61)
8       if(ent[2] == 0x75)
9         if(ent[3] == 0xde)
10          printf("fdata = %f\n" + *(unsigned int *)ent,
                ent->data.fdata); // crash
11   }
12 }
```

(a) Source code of the simplified sample.

```
1 %cmp31 = icmp eq i8 %9, 0x61
2 %cmp35 = icmp eq i8 %11, 0x75
3 %or.cond71 = and i1 %cmp31, %cmp35
4 %cmp39 = icmp eq i8 %5, 0xDE
5 %or.cond72 = and i1 %cmp39, %or.cond71
6 br i1 %or.cond72, label %if.then41, label %if.end58
```

(b) LLVM IR code of the nested if statements.



(c) Control flow graph of the assembly code.

Figure 2: Motivation example. Including the source code, the control flow graph of lines 6∼12 of the source code compiled with `-O3`, and the IR code of lines 7∼9 of the source code.

through all basic blocks at the LLVM IR (Intermediate Representation) level and inserts instrumentation codes at the beginning of each basic block.

B. Motivating Example

The coverage-feedback greybox fuzzers assume that the compilers or wrappers they use could carry out correct instrumentation during compilation, helping them to obtain accurate feedback from running states. However, it remains unexplored whether the compiler could guarantee the accuracy of the instrumentation. We will illustrate the problem through the simplified code snippet in Figure 2a.

The sample program[4] is simplified from a real-world code snippet of the Libxml2 library [41], which is a software library for parsing XML documents. It first parses the file header and compares the checksum with the magic number. After that, it copies the content of the file to `ent`. Then the program verifies the first four bytes in `ent` by a nested `if` structure one by one. Only if all the checks are passed through, it will trigger the crash at line 10.

Based on the intuition of coverage-based greybox fuzzing, every branch of the sample program should be instrumented so that AFL could find the crash easily. But it was surprising that for binary compiled with `-O3` (i.e., the default compiler optimization level that AFL uses), AFL could not find any crashes after 24 hours, and failed to cover line 8∼10. According to our analysis, this is because of the incomplete instrumentation. Figure 2c illustrates the control flow graph of the sample binary. As the graph shows, the nested `if` structure in Figure 2a (i.e., line 6∼9) contains four basic blocks (i.e., loc_400BB9, loc_400BEC, loc_400C13, and loc_400C1C). To achieve complete instrumentation and keep sensitive for all branches, the greybox fuzzer should instrument all four assembly-level basic blocks. However, an instrumentation error occurs while compiling the program in the LLVM mode of AFL. As Figure 2b shows, three compare statements (i.e., line 7∼9 in Figure 2a) are merged into one LLVM IR-level basic block because of the optimization that the compiler applies. Hence, `afl-clang-fast` only inserts one instrumentation at the beginning of the basic block. In the corresponding binary produced by the compiler, two assembly-level basic blocks (i.e., loc_400C13 and loc_400C1C) miss instrumentation because of the error. As a result, it loses the ability to perceive two missed branches, and triggers the vulnerability after these basic blocks with a low probability.

[4]https://groups.google.com/g/afl-users/c/e89ruXs7oOc/m/JjU03GlEBQAJ

C. Instrumentation errors of Greybox Fuzzers

According to our observation, there are mainly two types of errors in the instrumentation of coverage-based greybox fuzzers (we named them as "instrumentation errors" in this paper).

MIL (missed instrumentation location) means one basic block is missed by instrumentation. If the basic block is not instrumented, it will not give back some key information when it is executed, such as the coverage feedback [52], memory usage behavior [47], and so on. As a result, the fuzzer will not get useful feedback on this basic block and might discard some newly generated interesting samples, which could help the fuzzer explore more program states.

RIL (redundant instrumentation location) means that there is more than one instrumentation inserted in the same basic block. RIL increases the path depth, which misleads the fuzzers depend on the execution depth. In addition, because greybox fuzzers use fixed-size (e.g., 64KB) hash tables (i.e., bitmap) to store feedback information, RIL, which expands

Algorithm 1 The Detection Workflow.

Input: Instrumented *Program*
Output: Data about instrumentation errors of *Program*

```
1: P ← DISASSEMBLE(Program)
2: for bb = P.StartBB → P.EndBB do                          ▷ scan each basic block
3:     InstruSet ← ∅                          ▷ used to record all instrumentation in a bb
4:     offset ← 0  ▷ used to record the offset of instruction in the pattern sequence
5:     instSequence ← empty list              ▷ used to record successive instructions
6:     for inst = bb.StartInst → bb.EndInst do                  ▷ scan each instruction
7:         if INSTRUPATTERNMATCH(inst, offset) then
8:             instSequence.APPEND(inst)
9:             if offset == SIZE(intruPattern) − 1 then
10:                InstruSet ← InstruSet ∪ instSequence
11:                offset ← 0
12:                instSequence ← ∅
13:            else
14:                offset ← offset + 1
15:            end if
16:        end if
17:    end for
18:    if SIZE(InstruSet) == 0 then
19:        MILNum ← 1
20:    else if SIZE(InstruSet) > 1 then
21:        RILNum ← SIZE(InstruSet) − 1
22:    end if
23:    SAVEINSTRUINFO(bb, InstruSet, MILNum, RILNum)
24: end for
```

the number of instrumentation locations, could exacerbate bitmap collision.

Some researchers [21, 25, 26] notice that instrumentation could affect fuzzing, but few of them analyze and evaluate the impacts. As it is an underlying problem for almost all coverage-feedback greybox fuzzers, it motivates us to design an automatic tool to find and fix them to ensure fuzzing with correct instrumentation.

D. Focus of this paper

In this paper, we focus on studying the instrumentation errors of coverage-feedback greybox fuzzers with compile-time instrumentation. We try to develop a tool to find and fix these errors in the instrumented target binaries. Although we are not improving coverage-feedback greybox fuzzing from the fuzzing framework or strategies, our findings and tool can cooperate with existing fuzzers and benefit them as the instrumentation is accurate and complete.

III. METHOD

According to our analysis, instrumentation errors could cause incorrect coverage feedback and further harm fuzzing. To find and fix these errors, we firstly design a method to detect them in target binaries based on static analysis. Then we design two methods to fix instrumentation errors: firstly we try a straightforward method by changing compiler options, then we propose a general approach based on binary rewriting named InstruGuard. It should be noted that, in this section, the *basic block* represents the assembly-level basic block.

A. Detect Instrumentation Errors

To detect instrumentation errors, we disassemble the instrumented program and examine each basic block heuristically. As shown in Algorithm 1, for a basic block `bb` of the target program, we traverse all the instructions in it

Algorithm 2 The Repair Workflow.

Input: Instrumented *Program*
Output: Fixed *Program*

```
1: P ← DISASSEMBLE(Program)
2: for bb = P.StartBB → P.EndBB do
3:     InstruSet, MILNum, RILNum ← LOADINSTRUINFO(bb)
4:     if MILNum then
5:         INSERTINSTRU(bb)
6:     else if RILNum then
7:         for i = 0 → RILNum do
8:             DELETEINSTRU(bb, InstruSet[i])
9:         end for
10:    end if
11: end for
12: FixedProgram ← ASSEMBLE(P)
```

(line 6), leverage function `InstruPatternMatch` to check each instruction with the specific pattern of the instrumentation, which is implemented by the target fuzzer, and judge whether it belongs to the instrumentation (line 7). If so, we add the instruction `inst` to the instruction sequence `instSequence` (line 8). If the instrumentation pattern is matched exactly (line 9), we add `instSequence` to `InstruSet` (line 10) and reset the correlation variables (line 11∼12). When finishing the traversal, we check the number of instruction sequences in `InstruSet`. If there is no sequence in `InstruSet`, which means that `bb` has a MIL error, we set the `MILNum` to 1 (line 18∼19). If there is more than one sequence, this `bb` has one or more RIL errors, and we set the `RILNum` to the number of RIL errors (line 20∼21). After that, we save the information of the instrumentation set `InstruSet`, the number of MIL error `MILNum`, and the number of RIL errors `RILNum` (line 23).

```
1 mov   reg1, cs:__afl_area_ptr ;shared_mem
2 xor   reg2, cur_loc           ;edge_id
3 add   byte ptr [reg1+reg2], 1 ;shared_mem[edge_id]++
4 mov   reg2, cur_loc >> 1      ;reg2 stores the prev_loc
```

Listing 1: The template code of the instructions instrumented by `afl-clang-fast`. cur_loc is a constant and is generated during the compilation.

The patterns we used for matching are extracted from the instrumentation codes of fuzzers. Taking AFL as an example, the patterns are as following: ① for programs compiled with `afl-gcc`, we mark the `call` instruction to the record function `__afl_maybe_log`[5] as the feature of instrumentation; ② for programs compiled with `afl-clang-fast`, we highlight the instruction sequence of `xor`, `add/inc`, and `mov`, which represents the logic of the inserted instructions of instrumentation as shown in Listing 1. Specifically, AFL obtains the $edge\_id$ by applying XOR operation to the cur_loc and $prev\_loc$, adds one to $shared\_mem[edge\_id]$, and stores the left shifted cur_loc to $prev\_loc$. If InstruGuard finds the sequence, and the second argument of instructions `xor` (x_2) and `mov` (m_2) satisfy the equation: $m_2 = x_2 >> 1$, it marks the instruction sequence as the instrumentation. For

[5]AFL inserts a function call to `__afl_maybe_log` in each basic block, and the parameter to that call is a different value in each basic block. Therefore, when this instrumented code is executed, AFL can log which branch is triggered.

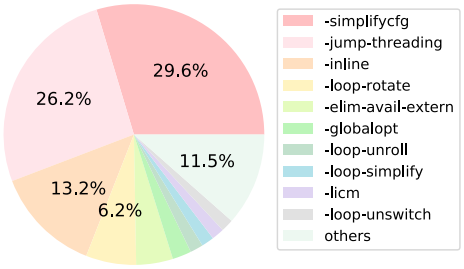


Figure 3: Distribution of the number of affected instrumentation location in different optimization flags of clang.

other fuzzers, we also manually analyze the features and use them as the prior knowledge for detection.

B. Fix Instrumentation Errors

1) Straightforward solution based on compiler options: As both of the instrumentation methods are implemented while compiling and the process are affected by compiler options, a straightforward solution is to control compiler optimization options. Intuitively, there should be no instrumentation errors if the basic blocks are not optimized during the compilation. We analyzed the impacts of the options on instrumentation locations by passing different optimization flags to the compiler. There are 27 flags of clang and 66 flags of gcc, which could change the number of instrumentation locations when used alone (Figure 3 displays the effect and marks the top 10 flags). We disabled all these flags to fix the instrumentation errors and checked the compiled binaries.

However, according to our experiments, it is not the proper way to fix instrumentation errors. Besides losing the performance advantage of compiler optimization, the instrumentation errors are not mitigated effectively by changing compiler options. As shown in Table III, the repair rate is only 49.86 % on average.

2) General solution based on Binary Rewriting: We propose an intuitive but more general repair method that directly fixes the instrumentation errors on the instrumented binaries based on binary rewriting. After identifying whether there are instrumentation errors and locating the position of the errors, we fix the MIL and RIL errors following the working procedure shown in Algorithm 2.

Firstly we disassemble the given program to the assembly code `P` (line 1). Then we traverse the basic blocks in `P` to fix the instrumentation errors (line 2). For a specific basic block `bb`, we load the result of Algorithm 1 to get instrumentation set `InstruSet`, the number of MIL errors `MILNum`, and the number of RIL errors `RILNum` (line 3). If this `bb` has a MIL error, we insert an instruction sequence (i.e., instrumentation) to the `bb` (line 4∼5). If this `bb` has one or more RIL errors, we remove all instrumentations except the last one in this `bb` (line 6∼9). Finally, we recompile the program `P` and get the repaired program (line 12).

Some instrumentation methods of greybox fuzzers, such as the compiler-level instrumentation of AFL, make the

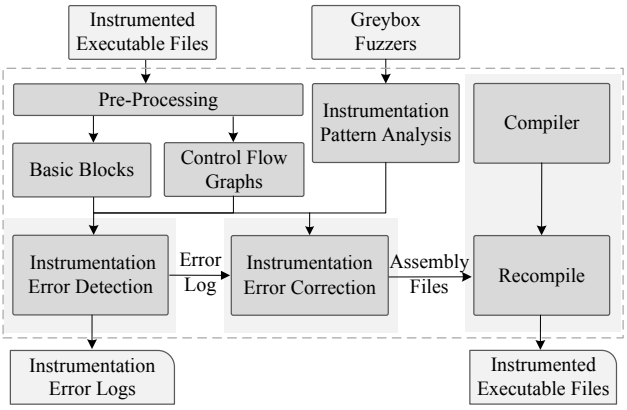


Figure 4: Architecture of InstruGuard.

instrumented code efficient via applying optimizations to instrumentations during the compilation (e.g., more efficient assign of registers). Hence, we rewrite the target program based on the instrumented binary instead of the vanilla binary and expect to retain the original correct instrumentations to keep the performance.

C. Implementation

We implement a framework named InstruGuard to find and repair instrumentation errors based on the above design. The architecture is shown in Figure 4. InstruGuard consists of three major components: the instrumentation error detection component, the error correction component and the compilation component. Before being processed by InstruGuard, the instrumented executable files are pre-processed to get the information of basic blocks and control flow graphs. The instrumentation patterns are also prepared as our prior knowledge.

The detection component detects instrumentation errors, which is developed based on IDApython [3]. It identifies instrumentation by matching the instruction sequence with particular patterns, and we adjust the matching patterns to detect instrumentation for different fuzzers as described in Section III-A. It is worth noting that IDA Pro sometimes identifies a basic block incorrectly due to a false branch label with no corresponding jump instruction. InstruGuard fixes it by checking the reference of each label with IDApython API `CodeRefsTo()`.

The error correction component is implemented based on RetroWrite [15], which is a precise and efficient binary-rewriting instrumentation tool. Firstly, we use it to disassemble the input binary file to the assembly code. Then InstruGuard modifies the assembly code to repair the instrumentation errors. For MIL errors, it inserts an instruction sequence, such as a function call to the record function (e.g., `__afl_maybe_log`) or a pattern sequence (e.g., as Listing 1 shows), to conduct instrumentation. For RIL errors, it comments out the redundant instructions.

The compilation component produces binary executables based on the modified assembly code. In detail, we write

a wrapper for the compiler (i.e., `gcc`) to recompile the program. According to our analysis in Section IV-B, `gcc` itself will not introduce new instrumentation errors except for several side-effects in `afl-gcc`. For example, AFL may miss the instrumentation after `.p2align` due to *Rule2* if it uses `afl-gcc` to compile programs under a UNIX-like operating system except for OpenBSD.

IV. Evaluation

In this section, we set experiments to answer the research questions raised in Section I. To answer RQ1, we instrumented real-world programs with the specific implementation of different coverage-based greybox fuzzers and tried different optimization levels of the compilers they use. Then we checked the instrumented programs with InstruGuard and analysed the root cause of the instrumentation errors. For RQ2 and RQ3, we repaired the instrumented programs with errors and calculated the fixed rate. Then we ran fuzzers with the original programs and the fixed ones, and compared the fuzzing results.

A. Setup Experiments

1) Program Dataset: We built a dataset of real-world programs by gathering 15 open-source Linux applications from recent papers published during the last two years with the corresponding version. The 15 applications are shown in Table I, including image parsing and processing libraries, text parsing tools, multimedia file processing libraries, and developing tools. In addition to version information, we also represent the default optimization option set by their developers and in which paper the application is selected as the test bench.

2) Instrumentation Methods: The instrumentation is related to fuzzers' implementation and compiler optimization. In our experiments, we selected 6 state-of-the-art fuzzers that get coverage feedback through instrumentation, i.e., AFL [52], FairFuzz [24], MOPT [27], MemLock [47], Angora [10] and AFL++ [16]. As AFL is the most popular coverage-based greybox fuzzer, we chose both the assembly-level mode (`afl-gcc`) and the compiler-level mode (`afl-clang-fast`) of it. FairFuzz, MOPT, and MemLock are three tools based on AFL, but towards different goals. We chose them to study the AFL's family. Chen et al. rewrite the algorithms of AFL in Angora, so we chose it as the comparison to avoid simple implementation bugs of greybox fuzzers. AFL++ is a union of several improvements of AFL and contains two different instrumentation methods, i.e., pcguard mode and LTO mode. We tested the above fuzzers and their instrumentation implementation to check whether instrumentation errors were common.

3) Fuzzing Setting: To study the impacts of instrumentation errors on fuzzing and test our solutions, we set fuzzing experiments with the instrumented program and the corresponding fuzzers. All the experiments were performed on five servers running Ubuntu 16.04.2 LTS and equipping with Intel(R) Xeon(R) CPU E5-2630 v3@2.40GHz (32 cores) and 32GB RAM. The compilers were gcc 5.4.0 and clang 6.0, as gcc 5.4.0 was the default gcc version of Ubuntu 16.04, and clang 6.0 was widely used by related works. For one target program, we ran experiments on the same server and configured it with the same seeds and command. We used the test cases of AFL as seeds that could be processed by the target application. Otherwise, we randomly selected files. The program arguments used in the evaluation were the same as the corresponding papers or issues. Each experiment timeout was set to 72 hours. Furthermore, we repeated all experiments 5 times and took the average value. We collected paths, coverage, and real bugs as the fuzzing results.

TABLE I: Real-world applications used in the experiment. `Paper` means where the application is selected as the test bench.

Package	Program	Version	Default Option	Paper
libwav	wav_gain	5cc8746	-O2	EnFuzz [13]
mp3gain	mp3gain	1.5.2	-O2	MOPT [27]
libjpeg	cjpeg	9a	-O2	EnFuzz
lupng	lupng	877a76f	-O0	EnFuzz
binutils	nm	2.29	-O2	CollAFL [18]
	objdump	2.29	-O2	PTrix [12]
	size	2.29	-O2	Angora [10]
	strip	2.29	-O2	Neuzz [36]
libming	listswf	0.4.8	-O2	ProFuzzer
ngiflib	gif2tga	c8488d5	-O	TortoiseFuzz
catdoc	catdoc	0.95	-O2	CollAFL
libpng	pngfix	1.6.34	-O2	Angora
libtiff	tiff2pdf	4.0.9	-O2	TortoiseFuzz
	tiff2ps	4.0.9	-O2	CollAFL
mpg321	mpg321	0.3.2	-O2	MOPT

B. Detect Instrumentation Errors (RQ1)

We compiled the programs in our dataset following the instructions of different fuzzers and with different compiler optimization levels (i.e., `O0` to `O3`, `O3` is the default option of AFL). Then we checked them with InstruGuard. The results are shown in Table II. Since FairFuzz and MOPT change the seed selection and mutation strategy without modifying the instrumentation method of the vanilla AFL, we got the same instrumentation results as AFL and did not put them in the table. We also did not list the data of the programs compiled by `afl-gcc` with `O0` and `O1` in the table since they almost had no instrumentation errors.

As Table II shows, instrumentation errors are common for different programs and different fuzzers. Programs compiled by different fuzzers all have instrumentation errors. Even small packages, like `libwav` whose total number of instrumentation locations is around 100, suffer from instrumentation errors. About one-fifth of libwav's basic blocks are incorrectly instrumented.

Instrumentation locations and the error rate vary a lot among different fuzzers. Programs produced by AFL with either the assembly-level or compiler-level instrumentation have the lowest instrumentation error rate. A deeper analysis shows that the majority of instrumentation errors caused by AFL are MIL errors. As for Memlock, it modifies the

TABLE II: Number of instrumentation locations (ILs) and the percentage of the instrumentation errors (Err-%) of packages compiled by different fuzzers and different optimization options. The symbol - means the corresponding fuzzer could not compile the package. `AFL(ASM)` column shows the results of binaries compiled by `afl-gcc`. `AFL` column shows the results of binaries compiled by `afl-clang-fast`. If not specified, we use `-O3` as the default option.

Program	AFL(ASM)		AFL-O0		AFL-O1		AFL-O2		AFL		Memlock		Angora		AFL++		AFL++-LTO	
	ILs	Err-%	ILs	Err-%	ILs	Err-%	ILs	Err-%	ILs	Err-%	ILs	Err-%	ILs	Err-%	ILs	Err-%	ILs	Err-%
catdoc	1,068	14%	1,071	8%	843	12%	915	13%	1,219	12%	1,098	8%	1,003	62%	660	60%	1,060	27%
libjpeg	5,488	49%	4,026	11%	3,240	13%	3,642	14%	4,532	15%	3,523	12%	3,328	65%	2,960	58%	4,906	31%
ngiflib	427	13%	392	8%	317	8%	422	8%	454	8%	407	8%	463	55%	253	56%	405	26%
libming	3,874	15%	3,749	16%	3,498	9%	5,321	11%	5,776	11%	3,075	37%	4,072	70%	-	-	3,461	27%
lupng	4,248	11%	3,329	11%	1,940	17%	2,464	18%	3,118	19%	3,708	11%	3,044	64%	1,588	63%	1,564	28%
mp3gain	3,441	12%	2,644	8%	1,740	18%	1,870	19%	2,428	19%	547	82%	2,335	63%	1,288	60%	2,141	28%
binutils	46,556	15%	43,098	9%	31,085	15%	37,653	15%	45,809	15%	61,718	10%	39,898	64%	31,953	64%	49,866	27%
libwav	101	24%	93	25%	72	19%	72	23%	92	23%	50	52%	70	63%	59	45%	45	35%
mpg321	2,287	16%	-	-	-	-	1,753	13%	1,773	14%	-	-	-	-	-	-	1,111	51%
libpng	16,868	12%	9,246	6%	7,332	15%	9,114	17%	9,417	21%	9,246	6%	7,916	65%	5,148	61%	3,955	52%
libtiff	25,086	13%	16,400	12%	12,851	14%	16,127	16%	16,692	20%	14,094	13%	11,844	63%	7,739	60%	9,411	52%
Average	9,949	18%	8,405	11%	6,292	14%	7,214	15%	8,301	16%	9,747	24%	7,397	63%	5,739	58%	7,084	35%

TABLE III: The number of the instrumentation errors before and after applying three fixing methods and the fixing rate. Ori-Err stands for the instrumentation errors that we found in the program compiled with vanilla fuzzer, and After-Err is the remaining errors after our fixes. The symbol - means the corresponding fuzzer could not compile the binary.

Program	Straightforward way			AFL(ASM)			AFL			Memlock		
	Ori-Err	After-Err	Fix Rate	Ori-Err	After-Err	Fix Rate	Ori-Err	After-Err	Fix Rate	Ori-Err	After-Err	Fix Rate
catdoc	186	106	43.01%	294	0	100%	186	0	100.00%	91	0	100.00%
cjpeg	520	86	83.46%	1,038	1	99.89%	520	0	100.00%	403	0	100.00%
gif2tga	41	21	48.78%	95	0	100%	41	0	100.00%	32	0	100%
listswf	688	749	-8.87%	1,059	0	100%	688	0	100.00%	1,416	1	100%
lupng	580	132	77.24%	452	0	100%	580	0	100.00%	362	1	99.72%
mp3gain	454	-	-	407	0	100%	454	0	100.00%	2,279	0	100.00%
nm	7,595	3,531	53.51%	7,729	15	99.81%	7,595	1	99.99%	4,088	1	99.98%
objdump	10,856	5,996	44.77%	10,789	39	99.64%	10,856	5	99.95%	6,524	1	99.98%
size	7,516	2,911	61.27%	7,667	19	99.75%	7,516	1	99.99%	4,084	1	99.98%
strip	8,881	-	-	7,649	14	99.82%	8,881	4	99.95%	4,638	1	99.98%
wav_gain	20	13	35.00%	24	0	100%	20	0	100.00%	53	0	100%
mpg321	223	-	-	356	0	100%	223	0	100%	-	-	-
pngfix	1,616	639	60.46%	2,079	1	99.95%	1,616	2	99.88%	597	0	100%
tiff2pdf	2,809	-	-	3,199	4	99.87%	2,809	16	99.43%	1,857	0	100%
tiff2ps	2,425	-	-	2,752	1	99.96%	2,425	4	99.84%	1,766	0	100%
Average			49.86%			99.91%			99.93%			99.96%

instrumentation method of the AFL to get more information about the memory and causes more instrumentation errors. On average, there exist instrumentation errors in more than one-fourth of basic blocks. With further analysis, the programs compiled by Memlock have more RIL errors than programs compiled by other fuzzers. Instrumentation errors are particularly common for programs compiled by Angora and AFL++'s pcguard mode. For almost every program compiled by Angora and AFL++, there are instrumentation errors in more than half of the basic blocks.

The results also show that instrumentation errors introduced by compiler-level instrumentation exist in all optimization options. In general, no matter what optimization option is set, more than 8% of the basic blocks of a program exist instrumentation errors. There are fewer instrumentation errors in the programs compiled by `afl-clang-fast` with `O0`. But for a specific program, compiling with `O0` could increase the instrumentation errors, such as `libwav` whose error rate reaches 25%.

We analyzed these instrumentation errors of different fuzzers to explore the root cause, and found out that the root causes are different for instrumentation errors introduced by assembly-level and compiler-level instrumentation. For assembly-level instrumentation, the errors are caused by side-effects of the implementation of `afl-gcc`. In detail, AFL misses the instrumentation after `.p2align` due to the *Rule2*. Besides, AFL adds redundant instrumentation code after labels that do not have the corresponding jump instruction (*Rule1*). For compiler-level instrumentation, the errors are caused by the transformation process from IR code to assembly code. As the example in Figure 2 shows, there is no MIL or RIL in the IR code, which is confirmed for other programs compiled with compiler-level instrumentation by checking their IR code. The errors happen during the transformation process from IR code to assembly code, the IR basic blocks will be split or merged due to the optimization, which causes the MIL or RIL.

AFL, Memlock, Angora, and AFL++ use compiler-level

TABLE IV: Code coverage, the number of real bugs, and the number of paths of the fuzzing result of the repaired program and the original program. The last line is average for coverage and paths, and sum for real bugs. `-re` stands for the result of the repaired program. `AFL(ASM)` column shows the results of binaries compiled by `afl-gcc`. `AFL` column shows the results of binaries compiled by `afl-clang-fast`.

Program	AFL(ASM)-O0			AFL(ASM)-O1			AFL(ASM)			AFL(ASM)-re			AFL			AFL-re		
	Bugs	Coverage	Paths	Bugs	Coverage	Paths	Bugs	Coverage	Paths	Bugs	Coverage	Paths	Bugs	Coverage	Paths	Bugs	Coverage	Paths
catdoc	1	50.0%	423	1	50.0%	702	1	50.0%	651	1	50.0%	684	1	50.0%	454	1	50.0%	453
cjpeg	2	14.3%	1,115	2	29.2%	1,581	2	29.5%	1,231	2	30.0%	1,199	2	29.2%	1,376	2	29.8%	671
gif2tga	4	75.3%	20,328	4	75.3%	18,247	3	75.3%	2,885	4	75.3%	42,635	2	75.3%	7,587	5	75.3%	40,453
listswf	3	18.0%	4,408	3	18.4%	5,207	3	18.4%	4,919	3	18.8%	5,625	3	21.0%	2,906	2	21.0%	2,430
lupng	0	38.7%	80	0	38.7%	75	0	38.7%	84	1	38.7%	92	0	38.7%	63	1	38.7%	71
mp3gain	5	57.3%	1,130	4	59.4%	1,341	5	59.5%	1,712	5	59.5%	1,685	6	58.8%	1,160	5	60.1%	1,118
nm	0	11.0%	2,267	0	10.5%	2,532	0	10.4%	2,563	0	10.0%	2,328	0	10.6%	2,497	0	10.9%	2,467
objdump	2	7.5%	1,980	2	7.6%	2,404	2	7.6%	2,505	2	7.6%	2,753	2	7.8%	2,190	2	7.5%	1,906
size	0	6.5%	1,818	0	6.4%	2,566	1	6.2%	2,933	0	6.8%	2,775	0	6.2%	1,690	0	6.0%	1,672
strip	1	7.9%	1,406	1	8.2%	1,688	0	8.1%	1,791	0	9.2%	2,619	0	8.0%	1,265	0	8.8%	1,716
wav_gain	2	77.0%	47	2	77.0%	40	2	77.0%	55	2	77.0%	58	2	77.0%	44	3	77.0%	46
mpg321	-	-	-	1	18.7%	186	1	18.6%	171	1	18.6%	177	1	18.6%	168	1	18.6%	183
pngfix	0	17.6%	334	0	17.7%	378	0	18.3%	324	0	18.4%	365	0	18.4%	336	0	18.4%	334
tiff2pdf	0	43.0%	5,025	0	42.8%	9,298	0	43.1%	9,376	0	43.2%	5,658	0	44.2%	5,526	0	44.3%	5,562
tiff2ps	0	33.1%	5,541	0	32.8%	6,711	0	36.6%	7,169	0	36.5%	5,941	0	37.4%	5,688	0	39.0%	5,288
Sum/Aver	20	32.7%	3,279	20	32.8%	3,530	20	33.2%	2,558	21	33.3%	4,973	19	33.4%	2,197	22	33.7%	4,291

Program	FairFuzz			FairFuzz-re			MOPT			MOPT-re			Memlock			Memlock-re		
	Bugs	Coverage	Paths	Bugs	Coverage	Paths	Bugs	Coverage	Paths	Bugs	Coverage	Paths	Bugs	Coverage	Paths	Bugs	Coverage	Paths
catdoc	1	50.0%	384	2	50.2%	385	1	50.1%	362	1	50.2%	417	1	50.0%	380	1	50.0%	385
cjpeg	2	30.3%	705	2	30.5%	715	3	30.9%	722	4	29.4%	632	0	26.1%	398	2	29.2%	648
gif2tga	3	75.3%	359	4	75.3%	695	3	75.3%	289	3	75.3%	316	2	79.4%	439	2	79.4%	391
listswf	3	64.9%	3,046	3	68.6%	3,699	0	68.6%	4,566	0	68.8%	4,640	3	66.1%	3,650	3	67.2%	4,102
lupng	1	38.7%	62	1	38.7%	67	0	38.7%	61	0	38.7%	68	0	38.7%	172	0	38.7%	171
mp3gain	6	60.7%	1,379	6	60.6%	1,390	5	57.7%	1,234	5	56.3%	1,225	5	60.8%	1,304	5	60.8%	1,315
nm	0	10.8%	3,928	0	10.0%	2,967	0	10.1%	2,015	0	10.1%	2,023	0	9.2%	5,808	0	9.4%	6,734
objdump	3	8.2%	3,029	3	8.7%	3,074	2	7.8%	2,187	2	8.2%	2,681	2	8.6%	3,397	2	8.3%	3,302
size	0	6.4%	1,973	1	6.9%	2,223	0	6.1%	1,974	0	7.0%	2,251	1	5.8%	2,929	0	5.7%	3,126
strip	0	8.4%	1,972	0	9.0%	2,279	0	9.1%	1,681	0	8.9%	1,840	0	8.1%	2,649	0	8.7%	2,744
wav_gain	2	73.1%	42	2	73.1%	43	1	73.1%	40	1	73.1%	40	1	73.1%	48	2	73.1%	56
mpg321	1	18.7%	192	1	18.7%	170	1	18.6%	191	1	18.6%	164	-	-	-	-	-	-
pngfix	0	17.7%	318	0	17.8%	304	0	17.7%	285	0	17.7%	288	0	18.3%	324	0	18.3%	362
tiff2pdf	0	44.5%	6,741	0	44.2%	6,624	0	39.4%	3,316	0	41.0%	4,556	0	36.1%	2,504	0	37.5%	2,661
tiff2ps	1	38.4%	6,062	1	38.4%	6,145	0	27.2%	1,514	0	27.6%	1,378	0	30.9%	2,165	0	31.2%	1,876
Sum/Aver	23	36.4%	2,013	26	36.7%	2,052	16	35.4%	1,362	17	35.4%	1,501	15	36.5%	1,869	17	37.0%	1,991

instrumentation, however, our experiments show that programs compiled by Angora and AFL++ have far more instrumentation errors. We did further research and found that besides the instrumentation, they do more modifications to the program during the compilation. Angora would split the basic block containing condition statements, such as `cmp` and `switch` statement, and generate two new branches. The two branches are the same in program logic and the branch target, but one of the branches will collect the information of the condition statement. This strategy is proposed to cut down the overhead of the constraint solving process. Similar to Angora, AFL++ employs pc-guard strategy to speed up the fuzzing, which adds new basic blocks into the program during the compilation process. These new basic blocks only contain a `mov` and a `jmp` instruction, which are used to load the edge id from the global memory.

As most of the coverage-based greybox fuzzers are based on AFL, and we have found great quantities of instrumentation errors caused by these state-of-the-art fuzzers, we can see that instrumentation errors are common in real-world coverage-based greybox fuzzers.

C. Repair Instrumentation Errors (RQ2)

We have demonstrated that there are vast numbers of MILs and RILs in programs compiled by fuzzers like AFL and Memlock. To repair the instrumentation errors, we applied two methods proposed in Section III-B to them. Table III shows the effect of our repairs. Since RetroWrite now could not handle binaries compiled by Angora and AFL++, we only listed 2 fuzzers in the table (More discussions in Section V).

As we mentioned in Section III-B1, we could not completely repair errors with the straightforward solution by controlling compiler optimization options. However, with InstruGuard, we almost eliminated all instrumentation errors with a rate of 99.93% for programs compiled by the compiler-level mode of AFL. For the programs compiled by the assembly-level mode of AFL and Memlock, InstruGuard achieved the similar effect, with a repair rate of over 99.9%.

We manually verified each unfixed instrumentation error in IDA pro by checking the assembly code of the fixed programs and did further research on them. We found they are detection errors instead of unfixed errors. Most of the errors are because of the missing branch label in the code

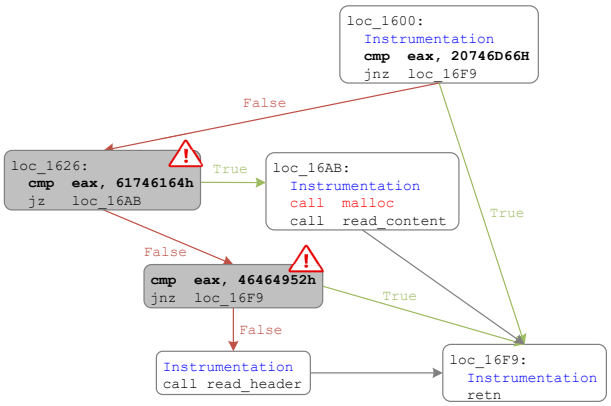


Figure 5: The simplified control flow graph of the code around the memory allocation point in `wav_gain`.

structures like `switch`. The argument of the jump instruction in the `switch` is a register like `rax` rather than a branch label. The original programs have jump table information so IDA Pro could identify the basic blocks accurately. However, after the re-compilation process of RetroWrite, the jump table information is lost, which makes IDA pro miss the label in the destination basic block of the jump instruction.

D. Fuzz with Repaired Instrumentation (RQ3)

To evaluate the effectiveness of our repairs, we ran a series of fuzzing experiments with the programs generated by fuzzers' instrumentation toolchains and the repair method. The results are shown in Table IV, and we use the number of real bugs, the line coverage of source code, and the number of paths as the metrics.

We can find that the fuzzing results of the repaired binaries are better at bugs and paths than the fuzzing results of the original ones with instrumentation errors. AFL finds 1 more real bugs in total and covers 0.1% more lines of code on average for the assembly-level mode. For compiler-level mode, AFL can trigger almost 3000 more paths, find 3 more real bugs, one of which is not reported before, and trigger similar coverage on average with the repaired programs. FairFuzz and Memlock are better at all three aspects with our binary rewrite solution, while MOPT seems not to benefit much. For specific programs, although AFL covers the same amount of code (75.3%) when fuzzing `gif2tga` compiled with `afl-clang-fast`, it finds more paths (from 7587 to 40453) with the repaired instrumentation, which results in finding more bugs (from 2 to 5). Memlock cannot find any bug in the origin `cjpeg`, but it finds 2 bugs in the repaired `cjpeg`.

New Vulnerability. With repaired instrumentation, we fortunately found a new vulnerability of memory leak in `wav_gain`, The original AFL could not find this bug in the program because of its inaccurate instrumentation. We analyzed the vulnerability based on the bug report and found there were several MIL errors, which are shown in Figure 5.

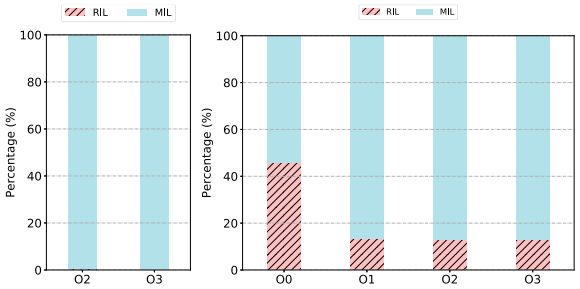
TABLE V: Fuzzing speed and the overview of p-values from Mann-Whitney U Test. The `Instru` column is the fuzzing speed of binaries compiled by `afl-clang-fast` with `O3` and fixed by InstruGuard. `AFL(ASM)` column shows the results of binaries compiled by `afl-gcc`. `AFL` column shows the results of binaries compiled by `afl-clang-fast`. `mpg321` can not be compiled with `-O0`.

Program	Instru	AFL(ASM)-O0		AFL(ASM)-O3		AFL	
	Means	Means	P-value	Means	P-value	Means	P-value
catdoc	455	404	0.500	760	0.037	539	0.201
cjpeg	136	162	0.265	204	0.105	280	0.147
gif2tga	955	506	0.104	727	0.265	712	0.148
listswf	206	112	0.338	377	0.265	455	0.265
lupng	2995	2162	0.047	1828	0.006	2683	0.500
mp3gain	66	59	0.338	60	0.500	173	0.030
nm	1170	1225	0.072	1463	0.030	1408	0.047
objdump	486	206	0.006	358	0.047	540	0.148
size	1147	1304	0.072	1265	0.265	1231	0.338
strip	301	100	0.047	241	0.500	333	0.265
wav_gain	153	158	0.417	112	0.047	131	0.104
mpg321	99	-	-	148	0.337	73	0.006
libpng	907	167	0.011	770	0.265	1400	0.500
tiff2pdf	667	389	0.202	377	0.202	655	0.500
tiff2ps	366	98	0.104	472	0.338	960	0.104
AVG	715	504	0.011	644	0.105	822	0.202

Just as the example in the Section II-B, the `switch` code is optimized to a series of comparisons and some basic blocks are not instrumented, so they cannot be perceived by AFL. We repaired the instrumentation errors along the vulnerable paths, giving us the ability to explore the vulnerable paths and discover the vulnerability. On the contrary, even though AFL might trigger the vulnerable path but would abandon the seed based on the wrong feedback caused by instrumentation errors. The new vulnerability is assigned with CVE-2020-28176 [1], which shows our repair is a benefit to fuzzing.

E. Performance & Overhead

To evaluate the performance of InstruGuard, we compared the execution speed of programs before and after being repaired while fuzzing process, besides compared with several compilation optimization levels. We took the execution speed of programs, which are compiled by `afl-clang-fast` with `O3` and fixed by InstruGuard, to represent the performance of InstruGuard, since `afl-clang-fast` is recommended by the fuzzing community and `O3` is the default optimization level. Firstly, we compared the performance of binaries fixed by InstruGuard with binaries compiled by `afl-gcc`. It is worth noting that binaries compiled by `afl-gcc` with `O0` and `O1` are free of instrumentation errors. As the performance results of binaries compiled with `O1` and with `O3` are similar, we did not show the result of `O1` in the table. As Table V shows, the average fuzzing speed (i.e., execution times each second) of binaries fixed by InstruGuard is 41.9% faster than binaries compiled by `afl-gcc` with `O0`, and 11.1% faster than `O3`. Compared to binaries compiled by `afl-gcc` with `O0`, 28.6% (4 of 14) of binaries fixed by InstruGuard are significantly faster (based on the p-values), and 20% (3 of 15) of binaries are faster compared to `O3`. Although binaries



(a) afl-gcc. **(b)** afl-clang-fast.

Figure 6: The distribution of MIL and RIL errors.

compiled by `afl-gcc` with `O0` are free from the instrumentation errors, our method could achieve better performance. For other optimization levels, binaries fixed by InstruGuard are also comparable in performance. Combining with Table IV, except InstruGuard, `O1` and `O0` might be good choices if one compiles binaries with `afl-gcc` and does not care about the execution speed while using AFL to test them. Their performance in finding bugs is comparable with `O3`, and they cause no instrumentation errors.

We also compared the performance of binaries fixed by InstruGuard with the unfixed binaries compiled by `afl-clang-fast` with `O3`. We found that binaries fixed by our tool are similar to them, regardless of the mean values or significant analysis. Surprisingly, the fuzzing speed of the fixed `mpg321` is even faster than the unfixed one, which might be due to the shrinking size of the binary after the rewriting. The shrunken parts comes from the relocation table, the `eh_frame_hdr` section, and the `eh_frame` section, which are discarded by RetroWrite after rewriting.

As for the overhead, the processing time of InstruGuard is 49 seconds on average for the tested programs, and the maximum time is 241 seconds for `mp3gain`. Compared with our entire fuzzing cycle (72 hours), the average overhead is minimum (0.02%).

F. Case Study

We took a further step based on the extended dataset to analyze the real-world instrumentation errors and shared some interesting observations. Figure 6 shows the distribution of the MIL and RIL errors. Since the programs compiled by `afl-gcc` with `O0` or `O1` are free from instrumentation errors, we did not present them in the figure.

1) MIL: Whether the program is compiled by `afl-gcc` or `afl-clang-fast`, MIL accounts for the majority of instrumentation errors. After preliminary manual analysis, we found that MIL occurs mostly in multiple continuous comparison logic in IR code. To verify this discovery, we wrote an LLVM pass to identify the corresponding logic and found that more than 70% of the MILs happen around the multiple continuous comparison logic. The pass also matched the multiple continuous comparison logic in IR code with the source code. The corresponding source code can be roughly

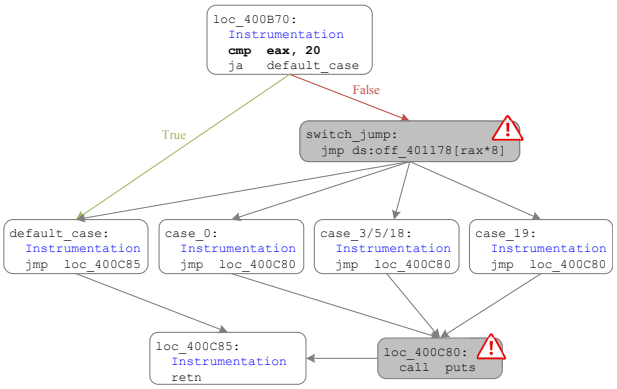


Figure 7: The simplified control flow graph of `switch` code structure in optimization option `O3` in `afl-clang-fast`.

categorized into 5 code patterns: ① `switch` statement, ② nested condition statements, such as `if-if`, ③ continuous `if-return` statement, ④ loop nested conditional statement, such as `for-if`, and ⑤ condition statement with multiple logical operators, such as `&&` or `||`.

Figure 7 is an example of `objdump` which contained two MILs. The switch_jump basic block and the other in loc_400C80 were missed with instrumentation by `afl-clang-fast` with `O3`. Without feedback from switch_jump basic block, AFL will treat two different paths, i.e., path with basic blocks of (loc_400B70, switch_jump, default_case) and the path of (loc_400B70, default_case), as the same and record only one of them. If AFL records path of (loc_400B70, default_case), it will generate fewer paths in which `eax` does not exceed 20. Under this situation, AFL will leave some numerically sensitive crashes undiscovered, and vice versa.

```
1 std::map<std::string,std::string> longs;
2 longs["--adjust" ] = "-a";
3 longs["--binary" ] = "-b";
4 ...
5 longs["--years" ] = "-Y"
```

Listing 2: The code fragment of `Params::getopt()`.

2) RIL: Most RILs exist in C++ programs for AFL. Listing 2 displays the function `Params::getopt()` in `exiv2`, which has the max number of RILs with the optimization option `O1`. Each assignment operation results in adding three more RILs during the compilation, and it has 120 RILs in total. Similar RIL errors happen in `bento4`, which contains 25 instrumentations in one basic block of function `AP4_HvccAtom::UpdateRawBytes()`. This basic block contains 24 call instructions which call the same function with different arguments, and after each call instruction there is a redundant instrumentation location.

V. Threat to Validity

Internal Validity. Fuzzing is a random process that may have an impact on the results of our evaluation. To mitigate

the effect of randomness, we extended the timeout to 72 hours and repeated our experiments 5 times according to the evaluation suggestions [20]. Besides, the process of binary rewriting could change the program behavior and introduce new vulnerabilities. In order to eliminate these possible effects, we double-checked each bug with original programs compiled with Address Sanitizer [33] while fuzzing, as well as gathered the line coverage of source code with the original programs.

External Validity. Due to the limitation of the RetroWrite, the binary rewriting tool we use, InstruGuard now could only find instrumentation errors for C++ programs and binaries compiled by Angora and AFL++, but could not fix them. It is mainly because that RetroWrite has trouble handling binaries that contain C++ exceptions, fails to disassemble binaries compiled by Angora, and generates non-compilable assembly code for binaries compiled by AFL++. However, since our repair method has been proven effective, once RetroWrite is updated or more powerful binary rewrite tools come out, InstruGuard will be able to fix instrumentation errors for all fuzzers.

In this paper, we also did not fix the program instrumented by fuzzers that use selective instrumentation. We selected 6 fuzzers as the research targets since they are representative and differ in instrumentation methods. They all try to explore as many paths as possible. However, other types of fuzzers also exist. For example, some fuzzers are designed for specific types of vulnerability, they could only instrument the "interesting" basic blocks or paths without triggering unrelated paths. For these fuzzers, we could extend InstruGuard with additional configurations to detect instrumentation errors along a specific path to ensure they act as expected.

VI. Related Work

The most related researches and techniques are presented in the following two parts, including greybox fuzzing and instrumentation.

Greybox fuzzing. Researchers improve fuzzing from various aspects. Some of them are applied after the instrumentation process. Vuzzer [32], Skyfire [19], Neuzz [36], and Faster Fuzzing [30] learn the important bytes or the grammars of the input files for more effective mutations. MOPT [27] optimizes the seed mutation scheduling strategies with the Particle Swarm Optimization (PSO) algorithm. AFLFast [5] allocates more energy to test the low-frequency paths to optimize the path exploration. Driller [38], QSYM [51] and T-fuzz [31] integrate static and dynamic analysis to prioritize hard-to-reach deeper paths.

Others modify the instrumented code to record more program behaviors or improve the sensitivity of the feedback. To guide the testing towards specific locations, AFLGo [6] modifies the instrumentation to calculate the distance between the current path and the target location. Memlock [47] and UAFL [42] collect the memory behavior of the programs to find more memory-related bugs. Angora [10] uses call stack while AFL-sensitive [43] uses n-gram to identify the different paths more specifically.

Instrumentation. Instrumentation approaches can be divided to binary instrumentation and source instrumentation. Usually binary instrumentation is applied when the source code of the tested program is unavailable, and could slow down the program significantly. Some fuzzers [31, 32, 38, 45] obtain feedback with dynamic binary instrumentation tools such as QEMU [4], DynamoRIO [7], and hardware-accelerated Intel Processor Trace. Fuzzers like AFL-Dyninst [40] use static instrumentation tools [8] to obtain feedback. Specifically, they use code patching techniques to inject callback events to gather coverage or other information.

AFL-family fuzzers [34, 39, 52] mainly get feedback through source instrumentation, which inserts a piece of specific code to each basic block while the target program is compiled with GCC or LLVM. Only a little research is conducted about the instrumentation problem. AFL-cc [37] leverages controlled optimization to minimize the difference (such as basic block layout) between the LLVM IR code and the binary, and tries to get accurate feedback for fuzzers. UNIFUZZ [26] notices that the instrumentation method might affect the program behavior, such as whether the certain bug could be triggered, and thus the fuzzing evaluation. However, they do not systemically analyze the impact of instrumentation errors and fix the problem.

VII. Conclusion

In this paper, we point out two types of instrumentation errors in coverage-based greybox fuzzers, and propose a framework named InstruGuard to find and fix instrumentation errors. We assessed the impacts of instrumentation errors on greybox fuzzers with a dataset of 15 real-world programs, and evaluated the effectiveness of our repairs through fuzzing from the aspect of the number of paths, line coverage of source code, and the number of real bugs. The results showed that instrumentation errors are common for most compilation optimization levels and coverage-based greybox fuzzers, and impact the fuzzing results. Our method fixed instrumentation errors effectively and benefited coverage-based greybox fuzzers.

Acknowledgement

We thank the anonymous reviewers of this work for their helpful feedback. This research is supported, in part, by National Natural Science Foundation of China (Grant No. U1936211, U1836117, U1836113, and 61902384), the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDC02020300. All opinions expressed in this paper are solely those of the authors.

References

[1] "CVE-2020-28176," https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-28176.
[2] "IDA Pro," https://www.hex-rays.com/products/ida/.
[3] "IDApython," https://www.hex-rays.com/products/ida/support/idapython_docs/.

[4] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005, pp. 41–41.

[5] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2016.

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[7] D. Bruening, E. Duesterwald, and S. Amarasinghe, "Design and implementation of a dynamic optimization framework for windows," in *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2000.

[8] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, 2000.

[9] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *ACM Conference on Computer and Communications Security*, 2018.

[10] P. Chen and H. Chen, "Angora: efficient fuzzing by principled search," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018.

[11] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1580–1596.

[12] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ser. Asia CCS '19, 2019.

[13] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[14] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 515–530.

[15] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1497–1511.

[16] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[17] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *29th USENIX Security Symposium*, 2020.

[18] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE, 2018.

[19] W. Junjie, C. Bihuan, W. Lei, and L. Yang, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy*, 2017, pp. 579–594.

[20] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[21] laf-intel, "Circumventing fuzzing roadblocks with compiler transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, [2019-1-27].

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

[23] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[24] C. Lemieux and K. Sen, "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[25] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017.

[26] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[27] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized Mutation Scheduling for Fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[28] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, 1990.

[29] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 787–802.

[30] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," *CoRR*, vol. abs/1711.02807, 2017.

[31] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy*, 2018, pp. 697–710.

[32] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 24th Network and Distributed System Security Symposium*, 2017.

[33] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: a fast address sanity checker," in *Usenix Conference on Technical Conference*, 2012.

[34] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," IEEE, 2016.

[35] ——, "Oss-fuzz - google's continuous fuzzing service for open source software," 2017.

[36] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[37] L. Simon and A. Verma, "Improving Fuzzing through Controlled Compilation," in *5th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[38] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the 23rd Network and Distributed Systems Security Symposium*, 2016.

[39] R. swiecki, "Honggfuzz," https://github.com/google/honggfuzz, 2016.

[40] talos vulndev, "AFL Dyninst," https://github.com/talos-vulndev/afl-dyninst, 2018.

[41] D. Veillard, "Libxml2: The XML C parser and toolkit of Gnome," http://www.xmlsoft.org, 2012.

[42] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 999–1010.

[43] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[44] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 2019, pp. 724–735.

[45] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[46] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *NDSS*, 2020.

[47] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE, 2020.

[48] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1643–1660.

[49] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "Pathafl: Path-coverage assisted fuzzing," *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[50] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 IEEE Symposium on Security*

and Privacy. IEEE, 2019, pp. 769–786.

[51] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 2018.

[52] M. Zalewski, "American fuzzy lop (AFL) fuzzer," http://lcamtuf.coredump.cx/afl, 2013.

[53] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *NDSS*, 2019.

[54] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th USENIX Security Symposium*, 2019, pp. 1099–1114.