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 and at different compiler optimization levels.

In this work, we propose InstruGuard, a tool for checking and fixing instrumentation errors in binary programs. Our tool is designed to be general and easy to use, making it a valuable addition to the field of greybox fuzzing. The results of our experiments show that InstruGuard is effective in fixing instrumentation errors and improving the performance of greybox fuzzers.
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 bug\(^1\) as the metrics to compare the results of original instrumented programs and the fixed ones with different fuzzers. The results of 72 hours × 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](https://github.com/Marsman1996/instruguard)

### 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 fuzzing (e.g., AFL [52]) attracts more attention recently because of its 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, the fuzzer 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. Next, we use AFL as the representation of coverage-based greybox fuzzing to introduce the details of instrumentation.

**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.

**Rule 1:** 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. **Rule 2:** If the line is in the section other than text\(^2\) or after .p2align they will leave this basic block un-instrumented even though the line satisfies the **Rule 1**.

**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

\(^1\)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.

\(^2\)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 == VTEPA) {
6   if (ent[0] == 0x66c) {
7     if (ent[1] == 0x61)
8     if (ent[2] == 0x75)
9     if (ent[3] == 0xde)
10       printf("fdata = %f\n", 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 %cmp39 = icmp eq i8 %9, 0x61
4 %cmp35 = icmp eq i8 %11, 0x75
5 %or.cond71 = 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.

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 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 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.

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 instrumented 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. So the fuzzer will not get useful feedback to keep the paths, including the basic block, lose the possibility to explore the following paths further.

RIL (redundant instrumented 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
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 compiler-level 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 (line 6), leverage function InstruPatternMatch to check each instruction with the specific pattern of the instrumentation and judge whether it belongs to the instrumentation (line 7). If so, we add the instruction inst to the 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).

Algorithm 1 The Detection Workflow.

<table>
<thead>
<tr>
<th>Input: Instrumented Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Data about instrumentation errors of Program</td>
</tr>
</tbody>
</table>

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
5:       ▷ scan each instruction
7:       if InstruPatternMatch(inst, offset) then
6:          instSequence.APPEND(inst)
7:       ▷ judge whether it belongs to the instrumentation
8:          offset ← 0
7:       else if offset == S ⊥ InstuSet[0] ⊥ 1 then
8:          instSequence ← ∅
7:          offset ← 0
8:       else
9:          if SIZE(InstruSet) == 0 then
9:             MILNum ← 1
10:         else if SIZE(InstruSet) > 1 then
11:             RILNum ← SIZE(InstruSet) − 1
12:         end if
13:     end if
14:     end for
15: end if
16: end for
17: if SIZE(InstruSet) == 0 then
18:    MILNum ← 1
19: else if SIZE(InstruSet) > 1 then
20:    RILNum ← SIZE(InstruSet) − 1
21: end if
22: end
23: SAVEMIInstrInfo(bb, InstruSet, MILNum, RILNum)
24: end for

Algorithm 2 The Repair Workflow.

<table>
<thead>
<tr>
<th>Input: Instrumented Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Fixed Program</td>
</tr>
</tbody>
</table>

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

The patterns we used for matching are extracted from the instrumented 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 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 (x2) and mov (m2) satisfy the equation: m2 = x2 >> 1, it marks the instruction sequence as the instrumentation. For

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

| mov | reg1, cs:_afl_area_ptr ; shared_mem |
| xor | reg2, cur_loc ; edge_id |
| add |byte ptr [reg1+reg2], 1 ; shared_mem[edge_id]++ |
| mov | reg2, cur_loc >> 1 ; reg2 stores the prev_loc'
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. Besides the performance differences of various options, 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 instrumented locations when used alone, and Figure 3 displays the top 10 flags of clang. We disabled all the 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).

We rewrite the instrumented binary instead of the vanilla binary because the compiler-level instrumentation is a higher-performance instrumentation mode. By keeping these instrumented locations, we control the run-time overhead within a reasonable range.

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. 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
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 as the comparison to avoid simple implementation bugs of AFL. We compiled the programs in our dataset following the instructions of different fuzzers and with different compiler optimization levels (i.e., $O_0$ to $O_3$ 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 00 and 01 in the table since they almost had no instrumentation errors.

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.

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., $O_0$ to $O_3$ 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 00 and 01 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 instrumented 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 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 pro-

<table>
<thead>
<tr>
<th>Package</th>
<th>Program</th>
<th>Version</th>
<th>Default Option</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>libwav</td>
<td>wav_gain</td>
<td>5cc874G6</td>
<td>-O2</td>
<td>EnFuzz [13]</td>
</tr>
<tr>
<td>libjpeg</td>
<td>jpeg</td>
<td>9a</td>
<td>-O2</td>
<td>EnFuzz</td>
</tr>
<tr>
<td>lupng</td>
<td>lpsngo</td>
<td>877a76f</td>
<td>-O0</td>
<td>EnFuzz</td>
</tr>
<tr>
<td>binutils</td>
<td>mn</td>
<td>2.29</td>
<td>-O2</td>
<td>MOPT</td>
</tr>
<tr>
<td>ngiflib</td>
<td>gifs2gta</td>
<td>c8488d5</td>
<td>-O3</td>
<td>ProFuzz</td>
</tr>
<tr>
<td>catdoc</td>
<td>catdoc</td>
<td>0.95</td>
<td>-O2</td>
<td>MemLock</td>
</tr>
<tr>
<td>libpng</td>
<td>pngfix</td>
<td>1.634</td>
<td>-O3</td>
<td>Angora</td>
</tr>
<tr>
<td>libtiff</td>
<td>tiff2pdf</td>
<td>4.09</td>
<td>-O2</td>
<td>TortoiseFuzz</td>
</tr>
<tr>
<td>mpg321</td>
<td>mpg321</td>
<td>0.32</td>
<td>-O2</td>
<td>MOPT</td>
</tr>
</tbody>
</table>
### TABLE II: Number of instrumented 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-clang-fast. AFL column shows the results of binaries compiled by afl-clang-fast. If not specified, we use -O3 as the default option.

<table>
<thead>
<tr>
<th>Program</th>
<th>AFL(ASM)</th>
<th>AFL-O0</th>
<th>AFL-O1</th>
<th>AFL-O2</th>
<th>AFL</th>
<th>Memlock</th>
<th>Angora</th>
<th>AFL++</th>
<th>AFL++-LTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ILs Err-%</td>
<td>ILs Err-%</td>
<td>ILs Err-%</td>
<td>ILs Err-%</td>
<td>ILs Err-%</td>
<td>ILs Err-%</td>
<td>ILs Err-%</td>
<td>ILs Err-%</td>
<td>ILs Err-%</td>
</tr>
<tr>
<td>catdoc</td>
<td>1,068 14%</td>
<td>1,071 8%</td>
<td>843 12%</td>
<td>915 13%</td>
<td>1,237 12%</td>
<td>1,098 8%</td>
<td>1,003 62%</td>
<td>660 60%</td>
<td>1,060 27%</td>
</tr>
<tr>
<td>libjpeg</td>
<td>5,488 49%</td>
<td>4,026 11%</td>
<td>3,240 13%</td>
<td>3,642 14%</td>
<td>4,653 15%</td>
<td>3,523 12%</td>
<td>3,328 65%</td>
<td>2,960 58%</td>
<td>4,906 31%</td>
</tr>
<tr>
<td>ngiflib</td>
<td>427 13%</td>
<td>392 8%</td>
<td>317 8%</td>
<td>422 8%</td>
<td>470 8%</td>
<td>407 8%</td>
<td>463 55%</td>
<td>253 56%</td>
<td>405 26%</td>
</tr>
<tr>
<td>libming</td>
<td>3,874 15%</td>
<td>3,749 16%</td>
<td>3,498 9%</td>
<td>5,321 11%</td>
<td>5,865 11%</td>
<td>3,075 37%</td>
<td>4,072 70%</td>
<td>- -</td>
<td>3,461 27%</td>
</tr>
<tr>
<td>luepg</td>
<td>4,248 11%</td>
<td>3,329 11%</td>
<td>1,940 17%</td>
<td>2,464 18%</td>
<td>3,173 19%</td>
<td>3,708 11%</td>
<td>3,044 64%</td>
<td>1,588 63%</td>
<td>1,564 28%</td>
</tr>
<tr>
<td>mp3gain</td>
<td>3,441 12%</td>
<td>2,644 8%</td>
<td>1,740 18%</td>
<td>1,870 19%</td>
<td>2,488 19%</td>
<td>547 82%</td>
<td>2,335 63%</td>
<td>1,288 60%</td>
<td>2,141 28%</td>
</tr>
<tr>
<td>bunittls</td>
<td>46,556 15%</td>
<td>43,098 9%</td>
<td>31,085 15%</td>
<td>37,653 15%</td>
<td>46,488 15%</td>
<td>61,718 10%</td>
<td>39,898 64%</td>
<td>31,953 64%</td>
<td>49,866 27%</td>
</tr>
<tr>
<td>libwav</td>
<td>101 24%</td>
<td>93 25%</td>
<td>72 19%</td>
<td>72 23%</td>
<td>93 23%</td>
<td>50 52%</td>
<td>70 63%</td>
<td>59 45%</td>
<td>45 35%</td>
</tr>
<tr>
<td>mpg321</td>
<td>2,287 16%</td>
<td>- -</td>
<td>- -</td>
<td>1,753 13%</td>
<td>1,550 14%</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
<td>- -</td>
</tr>
<tr>
<td>libpng</td>
<td>16,868 12%</td>
<td>9,246 6%</td>
<td>7,332 15%</td>
<td>9,114 17%</td>
<td>7,801 21%</td>
<td>9,246 6%</td>
<td>7,916 65%</td>
<td>5,148 61%</td>
<td>3,955 52%</td>
</tr>
<tr>
<td>libtiff</td>
<td>25,086 13%</td>
<td>16,400 12%</td>
<td>12,851 14%</td>
<td>16,127 16%</td>
<td>13,883 20%</td>
<td>14,094 13%</td>
<td>11,844 63%</td>
<td>7,739 60%</td>
<td>9,411 52%</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Straightforward way</th>
<th>AFL(ASM)</th>
<th>AFL</th>
<th>Memlock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ori-Err</td>
<td>After-Err</td>
<td>Fix Rate</td>
<td>Ori-Err</td>
</tr>
<tr>
<td>catdoc</td>
<td>186</td>
<td>106</td>
<td>43.01%</td>
<td>294</td>
</tr>
<tr>
<td>cjpeg</td>
<td>520</td>
<td>86</td>
<td>83.46%</td>
<td>1,038</td>
</tr>
<tr>
<td>gif2ga</td>
<td>41</td>
<td>21</td>
<td>48.78%</td>
<td>95</td>
</tr>
<tr>
<td>listswf</td>
<td>688</td>
<td>749</td>
<td>-8.87%</td>
<td>1,059</td>
</tr>
<tr>
<td>luepg</td>
<td>580</td>
<td>132</td>
<td>77.24%</td>
<td>452</td>
</tr>
<tr>
<td>mp3gain</td>
<td>454</td>
<td>- -</td>
<td>- -</td>
<td>407</td>
</tr>
<tr>
<td>mm</td>
<td>7,595</td>
<td>3,531</td>
<td>53.51%</td>
<td>7,729</td>
</tr>
<tr>
<td>objdump</td>
<td>10,856</td>
<td>5,996</td>
<td>44.77%</td>
<td>10,789</td>
</tr>
<tr>
<td>size</td>
<td>7,516</td>
<td>2,911</td>
<td>61.27%</td>
<td>7,667</td>
</tr>
<tr>
<td>strip</td>
<td>8,881</td>
<td>- -</td>
<td>- -</td>
<td>7,649</td>
</tr>
<tr>
<td>wav_gain</td>
<td>20</td>
<td>13</td>
<td>35.00%</td>
<td>24</td>
</tr>
<tr>
<td>mpg321</td>
<td>223</td>
<td>- -</td>
<td>- -</td>
<td>223</td>
</tr>
<tr>
<td>pngfix</td>
<td>1,616</td>
<td>639</td>
<td>60.46%</td>
<td>1,616</td>
</tr>
<tr>
<td>tiff2pdf</td>
<td>2,809</td>
<td>- -</td>
<td>- -</td>
<td>2,809</td>
</tr>
<tr>
<td>tiff2ps</td>
<td>2,425</td>
<td>- -</td>
<td>- -</td>
<td>2,752</td>
</tr>
</tbody>
</table>

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-clang-fast. 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. 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 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.
AFL(ASM)-O0 | AFL(ASM)-O1 | AFL(ASM) | AFL(ASM)-re | AFL | AFL-re
---|---|---|---|---|---
```
catdoc 1 50.0% 423 2 50.0% 702 1 50.0% 651 1 50.0% 684 1 50.0% 454 1 50.0% 453
cjpe 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
gif2ga 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% 64 0 38.7% 82 0 38.7% 63 0 38.7% 71
mp3gain 5 57.3% 1,114 4 59.4% 1,341 5 59.5% 1,712 5 59.8% 1,165 6 58.9% 1,118 5 58.8% 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
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
mpz32i - - - 1 18.7% 186 1 18.6% 171 1 18.6% 177 1 18.6% 168 1 18.6% 183
pgftxx 0 17.6% 334 0 17.7% 378 0 18.3% 324 0 18.4% 365 0 18.4% 334
tif2pdf 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
tif2ps 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
```

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-clang-fast.

As most of the coverage-based greybox fuzzers are based on AFL, and 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 unified 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 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.

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

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. 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, 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 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. Combined with Table IV except InstruGuard, O1 and O0 might be good choices if

<table>
<thead>
<tr>
<th>Program</th>
<th>Instru AFL(ASM)-O0</th>
<th>AFL(ASM)-O3</th>
<th>AFL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Means</td>
<td>P-value</td>
<td>Means</td>
</tr>
<tr>
<td>catdoc</td>
<td>455</td>
<td>0.500</td>
<td>760</td>
</tr>
<tr>
<td>cjpeg</td>
<td>136</td>
<td>0.265</td>
<td>204</td>
</tr>
<tr>
<td>gif2tga</td>
<td>955</td>
<td>0.104</td>
<td>727</td>
</tr>
<tr>
<td>listswf</td>
<td>206</td>
<td>0.338</td>
<td>377</td>
</tr>
<tr>
<td>lupng</td>
<td>2995</td>
<td>0.047</td>
<td>1828</td>
</tr>
<tr>
<td>mpg3gain</td>
<td>66</td>
<td>0.338</td>
<td>60</td>
</tr>
<tr>
<td>nm</td>
<td>1170</td>
<td>0.072</td>
<td>1463</td>
</tr>
<tr>
<td>objdump</td>
<td>486</td>
<td>0.006</td>
<td>358</td>
</tr>
<tr>
<td>size</td>
<td>1147</td>
<td>0.072</td>
<td>1265</td>
</tr>
<tr>
<td>strip</td>
<td>301</td>
<td>0.047</td>
<td>241</td>
</tr>
<tr>
<td>wav_gain</td>
<td>153</td>
<td>0.417</td>
<td>112</td>
</tr>
<tr>
<td>mpg321</td>
<td>99</td>
<td>-</td>
<td>148</td>
</tr>
<tr>
<td>libpng</td>
<td>907</td>
<td>0.011</td>
<td>770</td>
</tr>
<tr>
<td>tiff2pdf</td>
<td>667</td>
<td>0.202</td>
<td>377</td>
</tr>
<tr>
<td>tiff2ps</td>
<td>366</td>
<td>0.104</td>
<td>472</td>
</tr>
<tr>
<td><strong>AVG</strong></td>
<td>715</td>
<td>0.011</td>
<td>644</td>
</tr>
</tbody>
</table>
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 to 03, 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 03. 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 mp321 is even faster than the unfixed one, which might be due to the shrinking size of the binary after the rewriting. The shrunk 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 00 or 01 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 comparison logic in IR code. To verify this discovery, we wrote a loop nested conditional statement, such as if-if, if-return statement, if-return statement, if-return statement, and condition statement with multiple logical operators, such as & or 

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 01. 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 an redundant instrumented 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 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] tries to minimize the difference between the LLVM IR code and the binary by reducing the optimization. UNIFUZZ [26] notices that the instrumentation method might affect the fuzzing evaluation. However, they do not systemically analyze the impact and fix instrumentation errors.

**VII. CONCLUSION**

In this paper, we point out several 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 paths, line coverage, and real bugs. The results showed that instrumentation errors are common for all compilation optimizations and all coverage-based greybox fuzzers, and have a significant impact on 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. U1836117, U1836113, U1736209, 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**


