AFL^{2}oop: Loop Coverage Guided Greybox Fuzz Testing

Haochen Jin, Liwei Zheng, Zhanqi Cui*
Computer School, Beijing Information Science and Technology University, Beijing,China
Email:{jinhaochen0325, zlw, czq}@bistu.edu.cn

***Abstract*—Fuzz testing automatically generates and executes test cases, to detect more defects by covering more logical and state spaces of the program under test (PUT). However, it becomes more difficult to adequately test the PUT with increasing size and code complexity. Studies have shown that complex code is more likely to contain defects, and the loop is one of the main reasons for increased code complexity. Therefore, it is necessary to thoroughly test the loops, but existing fuzzers cannot focus on the loops of the PUT. To address this issue, we design a loop interval coverage metric to measure the testing adequacy of the loop. Additionally, we propose a greybox fuzz testing approach named AFL^{2}oop (AFL for Loop), which uses loop coverage as guidance. First, we analyze the loops of the PUT and expand the bitmap. Then, fuzz testing is guided by loop interval coverage and branch coverage. A prototype tool is implemented based on the proposed method, and experiments are carried out on four real-world software programs, such as LibXml2, LibMing, etc. The results show that AFL^{2}oop achieves higher coverage, triggers more crashes, and reproduces defects faster than AFL and FairFuzz.**

***Index Terms*—fuzz testing, coverage-based greybox fuzzing, loop analysis, crash reproduction, defect detection.**

I. Introduction

Software defects are faults, errors, or failures in software [1], which can result in unexpected outcomes [2] and potentially threaten people's safety or property in severe cases [3]. Software testing is used to detect potential defects in the program under test (PUT), but manually designing and executing test cases consumes much manpower and is inefficient. Fuzz testing [4] detects software defects by automatically or semi-automatically generating test inputs and monitoring the runtime status of the PUT. Fuzz testing, which reduces manual testing costs and improves testing efficiency, has become an effective method for detecting defects in real-world software [5]. Among them, greybox fuzz testing is not only more scalable but also combines the advantages of whitebox and blackbox fuzz testing [6], and has been widely studied in recent years [7] [8]. Typically, fuzzers with higher coverage can detect more defects in the PUT [6], because the coverage is closely related to the defect detection rate. Therefore, many greybox fuzzers try to increase the coverage of the logical and state space of the PUT as much as possible within a limited time.

As the size and code complexity of software increase, it becomes more difficult to achieve adequate testing coverage [9]. If limited resources are allocated equally to all the code, fuzz testing will be inefficient and unable to detect hidden defects in the PUT. Studies have shown that complex code is more likely to contain defects [10]. Loops cause the number of paths to increase exponentially in software, which is one of the main reasons for code complexity to increase. After using AFL (American Fuzzy Lop) [1] to test LibMing[2] and Libxml2[3] for 50 hours, we found that 17% and 50% of defects are located in loops. Testing loops is essential but challenging [11]. The number of loops increases with the code size and complexity of software, which presents a significant challenge for testing. Fuzzers may exhaust all testing resources before achieving high coverage [12]. In addition, existing fuzzers treat loops as simple branches, and existing coverage metrics are too general to evaluate the testing coverage of loops. As a result, it is difficult for fuzzers to adequately test loops in the PUT.

To address this issue, we design a loop interval coverage metric that divides the number of loop iterations into intervals to measure the coverage of the loop. Moreover, we propose an approach that uses loop coverage to guide greybox fuzz testing, which is called AFL^{2}oop (AFL for Loop). First, it analyzes the loop in PUT and extends the bitmap to collect the runtime coverage information. Then, it uses the loop interval coverage and branch coverage to guide the fuzz testing. Based on AFL^{2}oop, we have implemented a prototype tool and conducted experiments on four widely used real-world software. The experimental results show that AFL^{2}oop not only cover 2.9% and 1.8% more loop intervals than AFL and FairFuzz, but also triggers 33 and 67 more crashes than AFL and FairFuzz. In addition, AFL^{2}oop costs 4578 and 6089 seconds less than AFL and FairFuzz to trigger six defects, respectively.

In summary, this paper makes the following contributions:

- A loop interval coverage metric is designed to evaluate the coverage of the loops in the PUT. Based on the loop interval coverage metric, a loop coverage guided graybox fuzz testing approach is proposed.
- A prototype tool is implemented based on AFL^{2}oop and experiments are conducted to compare with AFL and

*Zhanqi Cui is the corresponding author.
DOI reference number: 10.18293/SEKE2023-075

[1] AFL. https://lcamtuf.coredump.cx/afl/
[2] LibMing. https://github.com/libming/libming
[3] Libxml2. https://gitlab.gnome.org/GNOME/libxml2

FairFuzz on four real-world software in terms of code coverage, defect detection and crash reproduction.

II. Loop coverage guided greybox fuzz testing

AFL^{2}oop consists of two phases: pre-processing and fuzz testing, as shown in Fig. 1. In the pre-processing phase, the PUT's basic blocks are analyzed, the bitmap and source code are instrumented. In fuzz testing, test cases are generated and executed, coverage is analyzed through basic blocks and loop coverage analysis. Interesting test cases are filtered and inserted into the test case queue. When the targets for coverage and number of crash triggers are reached, the testing is terminated and a report is generated.

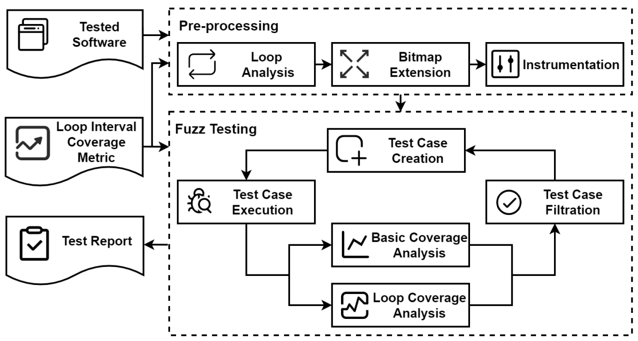

Fig. 1: Overview of AFL^{2}oop.

A. Loop interval coverage metric

Current coverage metrics for fuzz testing mainly measure the coverage of edges, branches, and lines of code in the PUT, but the coverage of loops is usually treated as a subset of branch coverage. This oversimplification limits the accuracy and effectiveness of existing coverage metrics in evaluating the quality and adequacy of fuzz testing results.

The branch condition of a loop determines the execution of statements contained in the loop body. If the condition is true, the statements are executed. Otherwise, the program will skip the loop body and continue with the statement next to the loop. However, using traditional coverage metrics that treat loops as branches and only record whether they are executed or not, as is commonly used in existing fuzz testing methods, ignores the variations due to different loop iteration counts. Instead, focusing on each individual iteration of the loop would be too resource-intensive. For example, in Listing. 1, when the loop iterates from 11 to 19 times, the same part of the loop is executed. Additionally, all the branch statements of the loop are executed if the loop iterates more than 20 times. However, a large number of loop iterations can cause an explosion of the solution space, leading to significant overhead in tracking loop coverage. To address this issue, a threshold should be set as the upper limit for counting iteration times during loop coverage analysis.

For this reason, we propose a new metric for loop interval coverage, called $LoopCover_k$, which uses k as the loop interval to evaluate loop coverage. In equation (1), L_{max} represents the maximum number of loop iterations counted for measuring loop coverage measurement. $R = \{R_1, R_2, \ldots, R_n, \ldots, R_m\}$ $(1 \leqslant n \leqslant m)$ is a set of loops, where R_n represents the n-th loop in the PUT. The set R_n is further defined as $R_n = \{r_{1,n}, r_{2,n}, \ldots, r_{i,n}, \ldots\}$ $(1 \leqslant i \leqslant \frac{L_{max}}{k})$, where $r_{i,n}$ indicates the coverage of the i-th loop interval. If a test case executes R_n l times, $(i-1)\cdot k < l \leqslant i \cdot k$, then the i-th loop interval of R_n is considered as covered, and $r_{i,n}$ is set to 1, otherwise it is set to 0.

Listing. 1: A code fragment of a loop

```
for (int i = 0; i < n; i ++) {
    if (i == 5)
        //...
    if (i == 10){
        //...
        if (n < 15)
            //...
    }
    if (i == 20)
        //...
}
```

B. Loop instrumentation and bitmap extension

Instrumentation involves inserting probes into the PUT without affecting its functionality. These probes collect run-time information during test case execution, which are used by the fuzzers to guide test case generation. For instance, AFL uses instrumentation to track edge coverage by inserting probes into basic blocks during compilation and dynamically computing coverage information of connected edges during test runs. This allows AFL to accumulate edge coverage information during testing.

The loop coverage information is used by AFL^{2}oop to guide its fuzz testing process. However, storing this information directly on the original shared bitmap of AFL can result in conflicts and errors. To address this issue, we implement an expansion of the shared bitmap from 64KB to 128KB, which was named *bit_loop_map*. Subsequently, the loops of the PUT are analyzed, and the instrumentation method of AFL is extended to collect the execution information generated by the test cases.

The procedure for analyzing and instrumenting loops in the PUT is outlined in Algorithm 1. The input of the algorithm is a set of functions (F). The algorithm begins by initializing three data structures: $loopMap$, *f_loop* and *f_branch* (lines 1-3). These data are used to record information about the loops, the precursor loops and the precursor branches, respectively. The algorithm then iterates over the basic blocks in F (lines 4-17). If a basic block BB under analyzed is located within a loop (line 5) and it has not been recorded in *bit_loop_map* the probe code is inserted into BB via $Instrumentation(BB)$ (line 7). Subsequently, the algorithm searches for the loops and branches to which BB belongs in *f_loop* and *f_branch*, otherwise using $null$ instead (lines 8-9). Next, the types of each basic blocks ($loop$, $branch$ and $normal$) need to be identified. The type of each basic block, along with the loop and branch to which it belongs is stored in $loopMap$, the loop and branch basic blocks are stored in *f_loop* and

$$LoopCover_k(R) = \frac{\sum_{n=1}^{m}(k \cdot R_n)/L_{max}}{m} = \frac{\sum_{n=1}^{m}(k \cdot \sum_{i=1}^{L_{max}/k} r_{i,n})/L_{max}}{m} \tag{1}$$

Algorithm 1. Analyze and instrument the PUT for loop coverage

```
Input: $F$ = the set of functions in the PUT.
Output: the PUT instrumented and basic block information of the loops
is stored in $loopMap$.
1: $loopMap$ = Hash()
2: $f\_loop$ = Stack()
3: $f\_branch$ = Stack()
4: for $BB$ in $F$:
5:     if isInLoop($BB$):
6:         if !$bit\_loop\_map$.full() and !$loopMap$.find($BB$):
7:             Instrumentation($BB$)
8:         $BBfLoop$ = $f\_loop$.findfLoop($BB$)
9:         $BBfbranch$ = $f\_branch$.findfBranch($BB$)
10:        if isLoopHeader($BB$) and !$LoopMap$.find($BB$):
11:            $loopMap$.add($BB$,"loop",$BBfLoop$,$BBfbranch$)
12:            $f\_loop$.push($BB$)
13:        else if (isIf($BB$) or isSwitch($BB$)) and $BB$.childNum $>$ 1:
14:            $loopMap$.add($BB$,"branch",$BBfLoop$,$BBfbranch$)
15:            $f\_branch$.push($BB$)
16:        else:
17:            $loopMap$.add($BB$,"normal",$BBfLoop$,$BBfbranch$)
18: $loopMap$.save()
```

f_branch (lines 10-17). Finally, $loopMap$ is stored in a file for subsequent analysis to guide fuzz testing (line 18).

C. Loop coverage guided greybox fuzz testing

The process of determining the interesting test cases according to their loop coverage of the PUT is outlined in Algorithm 2. The inputs of the algorithm include the coverage information of the loop interval and the basic blocks of loops ($testSetCover$), the result of the loop analysis performed during the pre-processing phase ($loopMap$), the loop interval (k), and the maximum number of loop iterations (L_{max}). First, the algorithm traverses all BBs in $loopMap$ (line 1). Then, if the type of BB is $loop$, the algorithm analyzes its coverage by using the number of loop iterations ($BB.bbCover$, from bit_loop_map) with the limitation of L_{max}, and k is also used to calculate the loop interval coverage of a test case (lines 2-7). If the loop interval in the $testSetCover$ has not been covered by another test case, current test case is considered to be interesting, and the algorithm returns $true$. If the type of BB is $branch$, identifying whether the branch has been covered by the test case need to examine all the successor basic blocks of the branch (lines 8-12). If the branch has not been covered in the $testSetCover$, the test case is considered to be interesting, and the algorithm returns $true$. Finally, if the test case fails to cover any new loop interval or branch, the algorithm returns $false$ (line 13). During fuzz testing, test cases that cover new edges are considered interesting and added to a queue, which is a circular linked list. The selected test case is executed one at a time and interesting test cases are added to the end of the queue. AFL^{2}oop's test case queue takes both loop interval and branch coverage into consideration, in addition to performance metrics.

Algorithm 2. Test case selection

```
Input: $testSetCover$, $loopMap$, $k$, and $L_{max}$.
Output: test case is interesting or not.
1: for $BB$ in $loopMap$:
2:     if $BB$.type is loop:
3:         $BB$.bbCover = min($BB$.bbCover , $L_{max}$)
4:         $interval$ = floor($BB$.bbCover / $k$)
5:             if !$testSetCover$.cover($interval$,$BB$):
6:                 $testSetCover$.setLoopCover($interval$,$BB$)
7:                 return true
8:     if $BB$.type is branch:
9:         for $child$ in $BB$:
10:            if !$child$.bbCover = 0 and !$testSetCover$.cover(0,$BB$):
11:                $testSetCover$.setBranchCover(0,$BB$)
12:                return true
13: return false
```

TABLE I: Statistics of the studied benchmarks

Projects	Version	Line of code	Count of loops	Count of Branches
LibMing[2]	0.4.8	238110	838	2822
LibXml2[3]	2.9.10	907769	1986	66764
mJS[4]	2.20.0	73381	145	3834
lrzip[5]	0.651	38308	690	5964
Total	-	1257568	3659	79384

III. EXPERIMENTS AND EVALUATIONS

A. EXPERIMENTAL DESIGN

1) Subjects & Benchmarks:

Subjects. A loop coverage guided greybox fuzzer is implemented based on AFL^{2}oop. Two existing greybox fuzzers, AFL and FairFuzz, are selected as subjects for comparison. AFL is an open-source greybox fuzzer that uses code coverage feedback to guide fuzz testing, while FairFuzz automatically adjusts input mutation to test rare parts of the PUT, achieving high coverage without sacrificing efficiency.

Benchmarks. The experiment uses four projects as benchmarks: LibXml2, mJS, LibMing, and lrzip, which cover various formats such as XML, JavaScript, Flash, and zip. These projects range in size from 3.8K to 90K lines of code and are commonly used in fuzzing research [13] [14]. The benchmarks and initial seeds are also used in a previous study. The total size of the five projects is 1.25 million LoC, containing 3659 loops and 79384 branches. The benchmark information is presented in TABLE I.

2) Evaluation Setups:

In experiments, we use three coverage, including branch coverage (BCOV), line coverage (LCOV) and $LoopCover_k$ to evaluate the coverage of the PUT by testing with different fuzzers. Additionally, we use the number of crash triggers and crash reproduction time to assess the ability of the fuzzers in detecting defects. BCOV measures the coverage of branches, while LCOV measures the coverage of lines of code in the

[4] mJS. https://github.com/cesanta/mjs
[5] lrzip. https://github.com/ckolivas/lrzip

TABLE II: $LoopCover_5$ of AFL^{2}oop, FairFuzz and AFL.

Programs	Fuzzers	$LoopCover_5$(%)
LibMing	AFL^{2}oop	96.9
	FairFuzz	96.3
	AFL	94.9
Libxml2	AFL^{2}oop	77.9
	FairFuzz	74.9
	AFL	73.6
mJS	AFL^{2}oop	80.9
	FairFuzz	78.6
	AFL	78.2
lrzip	AFL^{2}oop	62.1
	FairFuzz	60.8
	AFL	59.4

PUT. BCOV and LCOV are also widely used in other fuzz testing studies [3] [15]. The metric $LoopCover_k$, introduced in this paper is used to evaluate the coverage of the loop in the PUT, and L_{max} is set to 100 in the experiment. To collect loop execution information, an additional function is introduced into AFL and FairFuzz [5] without disrupting their core functionality. The experimental environments are as follows: 64-bit Ubuntu 18.04, with AMD Ryzen7 5800X @3.8 GHz CPU, and 64 GB RAM.

3) Research Questions:

To evaluate the performance of AFL^{2}oop, the following two research questions are addressed in the experiment:

- RQ1. How does AFL^{2}oop compare to other fuzzers in terms of code coverage?
- RQ2. How effective is AFL^{2}oop in detecting defects compared to other fuzzers?

B. Experimental Results and Discussions

1) ***RQ1****. How does AFL^{2}oop compare to other fuzzers in terms of code coverage:*

To evaluate the performance of AFL^{2}oop, we carry out an experiment to compare it with FairFuzz and AFL.

The results of the evaluation of the performance in terms of BCOV and LCOV are shown in Fig. 2 and Fig. 3, respectively. The results indicate that the BCOV and LCOV of the four PUTs increase over time when using different fuzzers. Among the fuzzers, AFL^{2}oop achieve the highest BCOV and LCOV when testing Libxml2, LibMing and lrzip, but the BCOV and LCOV of testing mJS by AFL^{2}oop are slightly lower than FairFuzz. Specifically, AFL^{2}oop's BCOV and LCOV are 0.8-4.3% and 0.1-0.8% greater than FairFuzz and AFL, respectively. However, FairFuzz achieves the greatest BCOV and LCOV when testing mJS. The results indicate that AFL^{2}oop can cover more branches and more lines of code in the PUTs by using loop coverage guided greybox fuzz testing.

In addition to evaluate the performance of AFL^{2}oop with BCOV and LCOV, we also compare its performance to AFL and FairFuzz with respect to $LoopCover_5$. The results of this comparison are shown in TABLE II. Overall, AFL^{2}oop achieves the highest $LoopCover_5$ among the four PUTs, with an average increase of 2.9% compare to AFL and 1.8% compare to FairFuzz. For example, in the case of Libxml2, AFL^{2}oop achieves 77.9% $LoopCover_5$, which is 3.0% and 5.3% greater than FairFuzz and AFL, respectively. This is due to the fact that AFL^{2}oop uses loop coverage guided greybox fuzz testing, test cases that cover new loop intervals are selected and inserted into the test case queue.

Answer for RQ1. The results indicate that AFL^{2}oop improves the performance of fuzz testing in terms of BCOV, LCOV and $LoopCover_5$ by utilizing loop coverage guided greybox fuzz testing. On average, AFL^{2}oop achieve 1.4% higher BCOV, 0.5% higher LCOV, and 2.9% higher $LoopCover_5$ compare to AFL. In comparison with FairFuzz, AFL^{2}oop achieve 0.5% higher BCOV, 0.1% higher LCOV, and 1.8% higher $LoopCover_5$.

2) ***RQ2****. How effective is AFL^{2}oop in detecting defects compared to other fuzzers:*

TABLE III gives the results of comparing the defects detection ability of AFL, FairFuzz and AFL^{2}oop. The experiment count the number of unique crashes triggered by the three fuzzers for the four PUTs. The results show that AFL^{2}oop outperforme both AFL and FairFuzz in terms of the number of unique crashes triggered. AFL^{2}oop triggers 483 unique crashes for the four PUTs, which is 33 more than AFL and 67 more than FairFuzz. AFL^{2}oop also triggers more unique crashes than AFL and FairFuzz for each individual PUT.

Additionally, we analyze the total number of defects detected by the three fuzzers in the four PUTs as well as the defects located in loops. Our analysis reveal that 29 defects are detected by the four PUTs, with 4 (17%) and 2 (50%) defects of LibMing and Libxml2 are located in loops, respectively. TABLE IV shows the time required for AFL, FairFuzz, and AFL^{2}oop to reproduce the six defects. AFL^{2}oop is able to reproduce four out of six defects in the shortest time, with a total time cost of 11557 seconds. AFL and FairFuzz can only reproduce one crash each in the shortest time, with total time costs of 16135 and 17646 seconds, respectively. For example, AFL^{2}oop reproduces the defect decompile.c:2015:37 of LibMing in 2629 seconds, which is 487 seconds faster than AFL, while FairFuzz is unable to detect and reproduce this crash. AFL^{2}oop also reproduce the defect valid.c:772:30 of Libxml2 in only 621 seconds, which is 2435 seconds and 564 seconds faster than AFL and FairFuzz, respectively.

Answer for RQ2. The experimental results show that AFL^{2}oop outperforms AFL and FairFuzz in terms of both the number of crashes triggered in the PUT and the time spent to reproducing crashes caused by defects in the loops. AFL^{2}oop trigger 483 crashes in the four PUTs, which is 33 more than AFL and 67 more than FairFuzz. In addition, AFL^{2}oop spent 39.6% and 68.2% less time than AFL and FairFuzz to reproduce the crashes caused by the defects located in the loops.

IV. DISCUSSIONS

A. Effects of loop interval

The results presented in Section III. indicate that AFL^{2}oop not only improves testing coverage but also detect more defects of the PUTs. The loop interval (k) is a crucial parameter of AFL^{2}oop. To assess the impact of k on AFL^{2}oop,

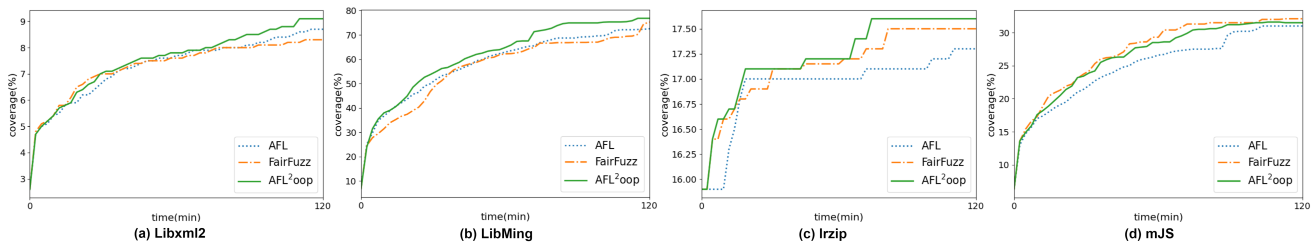

Fig. 2: BCOV(%) of AFL^{2}oop, FairFuzz and AFL.

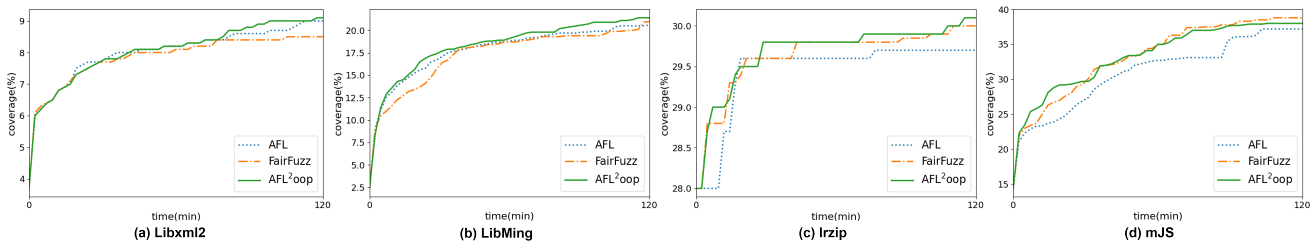

Fig. 3: LCOV(%) of AFL^{2}oop, FairFuzz and AFL.

TABLE III: The number of unique crashes triggered by AFL, FairFuzz and AFL^{2}oop.

Programs	AFL	FairFuzz	AFL^{2}oop
LibMing	349	319	360(+11)
Libxml2	100	87	118(+18)
mJS	1	10	5(-5)
lrzip	0	0	0(0)

TABLE IV: The time required for AFL, FairFuzz and AFL^{2}oop to reproduce the crashes of defects in the loops (in seconds).

Programs	Defects trigger crashes	AFL	FairFuzz	AFL^{2}oop
LibMing	outputscript.c:1440:17	2150	3098	1595(-555)
	decompile.c:2015:37	3116	T/O	2629(-487)
	decompile.c:1843:74	2722	3958	3044(+322)
	decompile.c:1843:64	3522	2385	2733(+348)
Libxml2	valid.c:772:30	3056	1185	621(-564)
	valid.c:729:30	1569	1762	935(-624)

experiments were conducted with k is set to 50, 20, 10, 5, and 1, respectively. The results show that despite the values of k are varied, both BCOV and LCOV of the four PUTs are increased with increasing of testing time. Among them, Libxml2, LibMing, and lrzip achieved the highest BCOV and LCOV when $k = 5$. The BCOV ($k = 5$) of mJS is slightly lower than $k = 100$, but the LCOV ($k = 5$) still exceeded that of all other k values.

As a result, loop interval for AFL^{2}oop is set as $k = 5$ in the experiments, as it leads to a higher coverage of branches and lines of both code compared to other values of k. Specifically, setting $k = 1$ generates numerous invalid test cases, while using excessively large intervals like $k = 50$ or $k = 20$ may miss interesting test cases and decrease coverage.

B. Threats to validity

Internal validity. The threats to internal validity mainly lies in the implementation of the studied fuzzers in our evaluation. To ensure accuracy, AFL was used as the basis for code instrumentation, bitmap extension, and fuzz testing. Moreover, the study used the same initial seed test cases as other studies to mitigate internal validity threats [13] [14]. The experiments were repeated five times and averaged to reduce the impact of randomly mutated test cases.

External validity. The threats to external validity mainly lie in the subjects and benchmarks. To ensure external validity, representative fuzzers AFL and FairFuzz [5] are used in the study, as they have been widely used and studied [13] [16]. Four widely-used programs of varying sizes and formats are used as benchmarks, including Flash, XML, JS, and zip. These benchmarks have been used in previous studies [13] [14] [17], ensuring that the results can be generalized to a wide range of software.

Construct validity. The threat to construct validity mainly lies in the metrics used in the experiments. Metrics such as branch coverage (BCOV), line of code coverage (LCOV), number of crashes triggered, time to reproduce crashes caused by defects located in the loop, and loop iteration interval coverage are used to evaluate the performance of fuzz testing tools. afl-cov[6] is used to obtain branch and line of code coverage, while the number of defect triggers and time to reproduce defects are evaluated using AFL ucrash and AddressSanitizer[7]. Loop iteration interval coverage is also proposed in the paper to evaluate the testing coverage of loop structures.

V. Related Work

Greybox fuzz testing, an integration of blackbox and whitebox fuzz testing, has gained significant attention as a practical and effective software testing method [18]. Greybox fuzzers use an evolutionary algorithm to generate new inputs and traverse paths in the program under test, guided by feedback information obtained from its execution. This approach can

[6] afl-cov. http://cipherdyne.com/afl-cov/

[7] AddressSanitizer. https://github.com/google/sanitizers /wiki/AddressSanitizer

be further categorized into coverage-guided and target-guided methods, depending on the approach used to guide the fuzzing process.

AFL and UnTracer [19] are coverage-based fuzz testing tools that collect execution information during testing to generate test cases based on coverage variants. UnTracer improves the coverage detection algorithm to enhance efficiency. Coverage-guided methods provide comprehensive coverage of the program within a specified test time frame, but may waste test resources on unimportant code.

AFLGo [14] and FairFuzz [5] are target-oriented fuzz testing tools that aim to cover target code blocks or functions. AFLGo calculates distance to the target block and adjusts the test case generation strategy, while FairFuzz prioritizes the exploration of less-visited parts of the program under test and adjusts the byte-level variation method.

Loop structures in code can lead to increased complexity and latent defects [20], but previous work in fuzz testing has not fully addressed the testing of these structures. Studies take loop complexity as an important component of complexity metrics in defect prediction models [21], and symbolic execution analyzed loop structures through techniques such as loop unwinding, invariant inference, and summarization [22]. However, machine learning-based defect prediction models still require manual confirmation and can result in a high false alarm rate, while symbolic execution methods can have limited scalability.

The proposed grey-box fuzz testing approach, AFL^{2}oop, focuses on the loop structure of the program under test as its target. It designs a loop interval coverage metric and derives its fuzz test by taking the coverage of the loop interval of the test case into consideration, ensuring adequate coverage of the loop structure while improving performance testing loop structure by dividing the loop into intervals.

VI. CONCLUSIONS

In this paper, we design the loop interval coverage metric as to measure the testing coverage of the loops in the PUT by fuzzers. Furthermore, we introduce AFL^{2}oop, which uses loop interval coverage metric to guide greybox fuzz testing. Based on the proposed approach, a prototype tool was implemented and compared with AFL and FairFuzz. The results show that AFL^{2}oop can cover more lines of code, branches, and loop intervals of the PUT. Furthermore, the number of crashes trigged and the efficiency of reproducing defects are also outperforms the other two fuzzers. In the future, we plan to encapsulate AFL^{2}oop as a plug-in to be integrated with other commonly used fuzzers to improve the testing efficiency and coverage of loops.

AKNOWLEDGEMENT

This work was supported in part by the Jiangsu Provincial Frontier Leading Technology Fundamental Research Project (BK20202001), the National Natural Science Foundation of China (No. 61702041), and the Beijing Information Science and Technology University "Qin-Xin Talent" Cultivation Project (No. QXTCP C201906).

REFERENCES

[1] P. Tripathy and K. Naik, *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[2] M. McDonald, R. Musson, and R. Smith, *The practical guide to defect prevention*. Microsoft Press, 2007.

[3] F. Gao, Y. Wang, L. Situ, and L. Wang, "Deep learning-based hybrid fuzz testing." *International Journal of Software & Informatics*, vol. 11, no. 3, 2021.

[4] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[5] 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, pp. 475–485.

[6] P. Wang, X. Zhou, K. Lu, T. Yue, and Y. Liu, "Sok: The progress, challenges, and perspectives of directed greybox fuzzing," *Challenges, and Perspectives of Directed Greybox Fuzzing*, 2020.

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

[8] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y.-C. Hu, "Hyperservice: Interoperability and programmability across heterogeneous blockchains," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 549–566.

[9] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux, "Precise identification of problems for structural test generation," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 611–620.

[10] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *Iet Software*, vol. 12, no. 3, pp. 161–175, 2018.

[11] T. Xie, L. Zhang, X. Xiao, Y.-F. Xiong, and D. Hao, "Cooperative software testing and analysis: Advances and challenges," *Journal of Computer Science and Technology*, vol. 29, no. 4, pp. 713–723, 2014.

[12] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 246–256.

[13] J.-M. Zhang, Z.-Q. Cui, X. Chen, H.-H. Wu, L.-W. Zheng, and J.-B. Liu, "Deltafuzz: Historical version information guided fuzz testing," *Journal of Computer Science and Technology*, vol. 37, no. 1, pp. 29–49, 2022.

[14] 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.

[15] 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 *USENIX Security Symposium*, 2021, pp. 2777–2794.

[16] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1634–1645.

[17] X. Gao, G. J. Duck, and A. Roychoudhury, "Scalable fuzzing of program binaries with e9afl," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1247–1251.

[18] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 499–513.

[19] 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.

[20] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 246–256.

[21] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *Iet Software*, vol. 12, no. 3, pp. 161–175, 2018.

[22] X. Xiaofei, "Static loop analysis and its applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1130–1132.