Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization

Yanhao Wang[1,2] Xiangkun Jia[3] Yuwei Liu[1,4] Kyle Zeng[5]
Tiffany Bao[5] Dinghao Wu[3] Purui Su[1,4,6]✉
[1]TCA/SKLCS, Institute of Software, Chinese Academy of Sciences [2]QiAnXin Technology Research Institute
[3]Pennsylvania State University [4]School of Cyber Security, University of Chinese Academy of Sciences
[5]Arizona State University [6]Cyberspace Security Research Center, Peng Cheng Laboratory
{wangyanhao, liuyuwei}@tca.iscas.ac.cn purui@iscas.ac.cn {tbao, zengyhkyle}@asu.edu {xxj56, duw12}@psu.edu

***Abstract*—Coverage-based fuzzing has been actively studied and widely adopted for finding vulnerabilities in real-world software applications. With coverage information, such as statement coverage and transition coverage, as the guidance of input mutation, coverage-based fuzzing can generate inputs that cover more code and thus find more vulnerabilities without prerequisite information such as input format. Current coverage-based fuzzing tools treat covered code equally. All inputs that contribute to new statements or transitions are kept for future mutation no matter what the statements or transitions are and how much they impact security. Although this design is reasonable from the perspective of software testing that aims at full code coverage, it is inefficient for vulnerability discovery since that 1) current techniques are still inadequate to reach full coverage within a reasonable amount of time, and that 2) we always want to discover vulnerabilities early so that it can be fixed promptly. Even worse, due to the non-discriminative code coverage treatment, current fuzzing tools suffer from recent anti-fuzzing techniques and become much less effective in finding vulnerabilities from programs enabled with anti-fuzzing schemes.**

To address the limitation caused by equal coverage, we propose *coverage accounting*, a novel approach that evaluates coverage by security impacts. Coverage accounting attributes edges by three metrics based on three different levels: function, loop and basic block. Based on the proposed metrics, we design a new scheme to prioritize fuzzing inputs and develop TortoiseFuzz, a greybox fuzzer for finding memory corruption vulnerabilities. We evaluated TortoiseFuzz on 30 real-world applications and compared it with 6 state-of-the-art greybox and hybrid fuzzers: AFL, AFLFast, FairFuzz, MOPT, QSYM, and Angora. Statistically, TortoiseFuzz found more vulnerabilities than 5 out of 6 fuzzers (AFL, AFLFast, FairFuzz, MOPT, and Angora), and it had a comparable result to QSYM yet only consumed around 2% of QSYM's memory usage on average. We also compared coverage accounting metrics with two other metrics, AFL-Sensitive and LEOPARD, and TortoiseFuzz performed significantly better than both metrics in finding vulnerabilities. Furthermore, we applied the coverage accounting metrics to QSYM and noticed that coverage accounting helps increase the number of discovered vulnerabilities by 28.6% on average. TortoiseFuzz found 20 zero-day vulnerabilities with 15 confirmed with CVE identifications.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24422
www.ndss-symposium.org

I. Introduction

Fuzzing has been extensively used to find real-world software vulnerabilities. Companies such as Google and Apple have deployed fuzzing tools to discover vulnerabilities, and researchers have proposed various fuzzing techniques [4, 6, 7, 12, 18, 29, 33, 43, 45, 47, 56, 60, 61]. Specifically, coverage-guided fuzzing [4, 6, 12, 18, 33, 43, 45, 56, 61] has been actively studied in recent years. In contrast to generational fuzzing, which generates inputs based on given format specifications [2, 3, 16], coverage-guided fuzzing does not require knowledge such as input format or program specifications. Instead, coverage-guided fuzzing mutates inputs randomly and uses coverage to select and prioritize mutated inputs.

AFL [60] leverages edge coverage (a.k.a. branch coverage or transition coverage), and libFuzzer [47] supports both edge and block coverage. Specifically, AFL saves *all* inputs with new edge coverage, and it prioritizes inputs by size and latency while guaranteeing that the prioritized inputs cover all edges. Based on AFL, recent work advances the edge coverage metrics by adding finer-grained information such as call context [12], memory access addresses, and more preceding basic blocks [53].

However, previous work treats edges equally, neglecting that the likelihoods of edge destinations being vulnerable are different. As a result, for all the inputs that lead to new coverage, those that execute the newly explored code that is less likely to be vulnerable are treated as important as the others and are selected for mutation and fuzzing.

Although such design is reasonable for program testing which aims at full program coverage, it delays the discovery of a vulnerability. VUzzer [43] mitigates the issue by de-prioritizing inputs that lead to error-handling code, but it depends on taint analysis and thus is expensive. CollAFL [18] proposes alternative input prioritization algorithms regarding the execution path, but it cannot guarantee that prioritized inputs cover all security-sensitive edges and it may cause the fuzzer to be trapped in a small part of the code. AFL-Sensitive [53] and Angora [12] add more metrics complementary to edges, but edges are still considered equally, so the issue still exists for the inputs with the same value in the complementary metrics. LEOPARD [15] considers function coverage instead of edges and it weights functions differently, but it requires static analysis to preprocess, which causes extra performance overhead. Even worse, these approaches are

all vulnerable to anti-fuzzing techniques [23, 28] (See II-D). Therefore, we need a new input prioritization method that finds more vulnerabilities and is less affected by anti-fuzzing techniques.

In this paper, we propose *coverage accounting*, a new approach for input prioritization. Our insight is that, any work that adds additional information to edge representation will not be able to defeat anti-fuzzing since the fundamental issue is that current edge-guided fuzzers treat coverage equally. Moreover, memory corruption vulnerabilities are closely related to sensitive memory operations, and sensitive memory operations can be represented at different granularity of function, loop, and basic block [27]. To find memory corruption vulnerabilities effectively, we should cover and focus on edges associated with sensitive memory operations only. Based on this observation, our approach assesses edges from function, loop, and basic block levels, and it labels edges security-sensitive based on the three metrics of different levels. We prioritize inputs by new security-sensitive coverage, and cull the prioritized inputs by the hit count of security-sensitive edges and meanwhile guarantee the selected inputs cover all visited security-sensitive edges.

Based on the proposed approach, we develop TortoiseFuzz, a greybox coverage-guided fuzzer[1]. TortoiseFuzz does not rely on taint analysis or symbolic execution; the only addition to AFL is the coverage accounting scheme inserted in the step of queue culling (See II).

TortoiseFuzz is simple yet powerful in finding vulnerabilities. We evaluated TortoiseFuzz on 30 popular real-world applications and compared TortoiseFuzz with 6 state-of-the-art greybox [7, 31, 36, 60] and hybrid fuzzers [12, 59]. We calculated the number of discovered vulnerabilities, and conducted Mann-Whitney U test to justify statistical significance between TortoiseFuzz and the compared fuzzers. TortoiseFuzz performed better than 5 out of 6 fuzzers (AFL, AFLFast, FairFuzz, MOPT, and Angora), and it had a comparable result to QSYM yet only consumed, on average, around 2% of the memory resourced used by QSYM. TortoiseFuzz found 20 zero-day vulnerabilities with 15 confirmed with CVE identifications.

We also compared coverage accounting metrics against AFL-Sensitive and LEOPARD, and the experiment showed that our coverage accounting metrics performed significantly better in finding vulnerabilities than both metrics. Furthermore, we applied the coverage accounting metrics to QSYM, and we noticed that coverage accounting boosted the number of discovered vulnerabilities by 28.6% on average.

To foster future research, we will release the prototype of TortoiseFuzz open-sourced at https://github.com/TortoiseFuzz/ as well as the experiment data for reproducibility.

Contribution. In summary, this paper makes the following contributions.

- We propose *coverage accounting*, a novel approach for input prioritization with metrics that evaluates edges in terms of the relevance of memory corruption vulnerabilities. Our approach is lightweighted without expensive analyses, such as taint analysis and symbolic execution, and is less affected by anti-fuzzing techniques.
- We design and develop TortoiseFuzz, a greybox fuzzer based on coverage accounting. We will release TortoiseFuzz with source code.
- We evaluated TortoiseFuzz on 30 real-world programs and compared it with 4 greybox fuzzers and 2 hybrid fuzzers. As a greybox fuzzer, TortoiseFuzz outperformed all the 4 greybox fuzzers and 1 hybrid fuzzers. TortoiseFuzz achieved a comparable result to the other hybrid fuzzer, QSYM, yet only spent 2% of the memory resource costed by QSYM. TortoiseFuzz also found 20 zero-day vulnerabilities, with 15 confirmed with CVE IDs.

[1]The name comes from a story of Aesop's Fables. A tortoise is ridiculed by a rabbit at first in a race, crawls slowly but steadily, and beats the rabbit finally. As American Fuzzy Lop is a kind of rabbit, TortoiseFuzz wins.

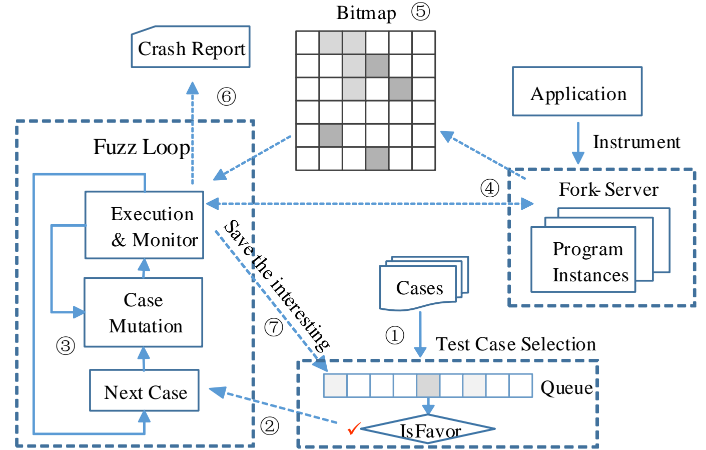

Fig. 1: The framework of AFL.

II. Background

In this section, we present the background of coverage-guided fuzzing techniques. We first introduce the high-level design of coverage-guided fuzzing, and then explain the details of input prioritization and input mutation in fuzzing.

A. Coverage-guided Fuzzing

Fuzzing is an automatic program testing technique for generating and testing inputs to find software vulnerabilities [38]. It is flexible and easy to apply to different programs, as it does not require the understanding of programs, nor manual generation of testing cases.

At a high level, coverage-guided fuzzing takes an initial input (seed) and a target program as input, and produces inputs triggering program error as outputs. It works in a loop, where it repeats the process of selecting an input, running the target program with the input, and generating new inputs based on the current input and its running result. In this loop, coverage is used as the fundamental metric to select inputs, which is the reason why such techniques are called coverage-guided fuzzing.

Figure 1 shows the architecture of AFL [60], a reputable coverage-guided fuzzer based on which many other fuzzers

are developed. AFL first reads all the initial seeds and moves them to a testcase queue (①), and then gets a sample from the queue (②). For each sample, AFL mutates it with different strategies (③) and sends the mutated samples to a forked server where the testing program will be executed with every mutated sample (④). During the execution, the fuzzer collects coverage information and saves the information in a global data structure. AFL uses edge coverage, which is represented by a concatenation of the unique IDs for source and destination basic blocks, and the global data structure is a bitmap (⑤). If the testing program crashes, the fuzzer marks and reports it as a proof of concept of a vulnerability (⑥). If the sample is interesting under the metrics, the fuzzer puts it into the queue and labels it as "favored" if it satisfies the condition of being favored (⑦).

B. Input Prioritization

Input prioritization is to select inputs for future mutation and fuzzing. Coverage-guided fuzzers leverage the coverage information associated with the executions to select inputs. Different fuzzers apply different criteria for testing coverage, including block coverage, edge coverage, and path coverage. Comparing to block coverage, edge coverage is more delicate and sensitive as it takes into account the transition between blocks. It is also more scalable than path coverage as it avoids path explosion.

AFL and its descendants use edge coverage for input prioritization. In particular, AFL's input prioritization is composed of two parts: input filtering (step ⑦ in Figure 1) and queue culling (step ① in Figure 1). Input filtering is to filter out inputs that are not "interesting", which is represented by edge coverage and hit counts. Queue culling is to rank the saved inputs for future mutation and fuzzing. Queue culling does not discard yet re-organizes inputs. The inputs with lower ranks will have less chance to be selected for fuzzing. Input filtering happens along with each input execution. Queue culling, on the other hand, happens after a certain number of input executions which is controlled by mutation energy.

1) Input Filtering

AFL keeps a new input if the input satisfies either of the following conditions:

- The new input produces new edges between basic blocks.
- The hit count of an existing edge achieves a new scale.

Both conditions require the representation of edge. To balance between effectiveness and efficiency, AFL represents an edge of two basic blocks by combining the IDs of the source and destination basic blocks by shift and xor operations.

```
cur_location = <COMPILE_TIME_RANDOM>;
bitmap[cur_location ⊕ prev_location]++;
prev_location = cur_location » 1;
```

For each edge, AFL records whether it is visited, as well as the times of visit for each previous execution. AFL defines multiple ranges for the times of visit (i.e., bucketing). Once the times of visit of the current input achieves a new range, AFL will update the record and keep the input.

The data structure for such record is a hash map, and thus is vulnerable for the hash collision. CollAFL [18] points out a new scheme that mitigates the hash collision issue, which is complementary to our proposed approach for input prioritization.

2) Queue Culling

The goal of queue culling is to concise the inputs while maintaining the same amount of edge coverage. Inputs that remained from the input filtering process may be repetitive in terms of edge coverage. In this process, AFL selects a subset of inputs that are more efficient than other inputs while still cover all edges that are already visited by all inputs.

Specifically, AFL prefers inputs with less size and less execution latency. To this end, AFL will first mark all edges as not covered. In the next, AFL iteratively selects an edge that is not covered, chooses the input that covers the edge and meanwhile has the smallest size and execution latency (which is represented as a score proportional to these two elements), and marks all edges that the input visits as covered. AFL repeats this process until all edges are marked as covered.

Note that in AFL's implementation, finding the best input for each edge occurs in input filtering rather than in queue culling. AFL uses a map `top-rate` with edges as keys and inputs as values to maintain the best input for each edge. In the process of input filtering, if AFL decides to keep an input, it will calculate the score proportional to size and execution time, and update the `top-rate`. For each edge along with the input's execution path, if its associated input in `top-rate` is not as good as the current input in terms of size and execution time, AFL will replace the value of the edge with the current input. This is just for ease of implementation: in this way, AFL does not need a separate data structure to store the kept inputs in the current energy cycle with their size and latency. For the details of the algorithm, please refer to Algorithm 1 in Section IV.

3) Advanced Input Prioritization Approaches

Edge coverage, although well balances between code coverage and path coverage, is insufficient for input prioritization because it does not consider the finer-grained context. Under such circumstances, previous work proposes to include more information to coverage representation. Angora [12] proposes to add a calling stack, and AFL-Sensitive [53] presents multiple additional information such as memory access address (memory-access-aware branch coverage) and n-basic block execution path (n-gram branch coverage).

This advancement improves typical edge coverage to be finer-grained, but it still suffers from the problem that inputs may fall into a "cold" part of a program which is less likely to have memory corruption vulnerabilities yet contributes to new coverage. For example, error-handling codes typically do not contain vulnerabilities, and thus fuzzer should avoid to spend overdue efforts in fuzzing around error-handling code. VUzzer [43] de-prioritizes the inputs that lead to error handling codes or frequent paths. However, it requires extra heavyweight work to identify error-handling codes, which makes fuzzing less efficient.

CollAFL [18] proposes new metrics that are directly related to the entire execution path rather than single or

a couple of edges. Instead of queue culling, it takes the total number of instructions with memory access as metrics for input prioritization. However, CollAFL cannot guarantee that the prioritized inputs cover all the visited edges. As a consequence, it may fall into a code snippet that involves intensive memory operations yet is not vulnerable, e.g., a loop with a string assignment.

LEOPARD [15] keeps queue culling yet add an additional step, prioritizing the selected inputs from queue culling by a function-level coverage metrics, rather than choosing randomly in AFL. The approach is able to cover all visited edge in each fuzzing loop, but it requires to preprocess the targeting programs for function complexity analysis and thus brings performance overhead.

C. *Input Mutation and Energy Assignment*

Generally, input mutation can also be viewed as input prioritization: if we see the input space as all the combinations of bytes, then input mutation prioritizes a subset of inputs from the input space by mutation. Previous work design comprehensive mutation strategies [12, 26, 31, 58] and optimal mutation scheduling approaches [36]. These input mutation approaches are all complementary to our proposed input prioritization scheme.

Similarly, energy assignment approaches such as AFLFast [7], AFLGo [6], FairFuzz [31] also prioritizes inputs by deciding the number of children inputs mutated from a father input. AFLFast [7] assigns more energy to the seeds with low frequency based on the Markov chain model of transition probability. While AFLGo [6] becomes a directed fuzzer, which allocates more energy on targeted vulnerable code. FairFuzz [31] marks branches that are hit fewer times than a pre-defined rarity-cutoff value as rare branches, and optimizes the distribution of fuzzing energy to produce inputs to hit a given rare branch.

D. *Anti-Fuzzing Techniques*

Current anti-fuzzing techniques [23, 28] defeat coverage-guided fuzzers by two design deficiencies: 1) most coverage-guided fuzzers do not differentiate the coverage of different edges, and 2) hybrid fuzzers use heavyweight taint analysis or symbolic execution. Anti-fuzzing techniques deceive fuzzers by inserting fake paths, adding a delay in error-handling code, and obfuscating codes to slow down dynamic analyses.

Current anti-fuzzing techniques make coverage-guided fuzzers much less effective in vulnerability discovery, causing 85%+ performance decrease in exploring paths. Unfortunately, many of the presented edge-coverage-based fuzzers [12, 15, 43, 53, 60] suffer from the current anti-fuzzing techniques. VUzzer is affected due to the use of concolic execution. LEOPARD, which considers function-level code complexity as a metric for input prioritization, is vulnerable to fake paths insertion. As path insertion increases the complexity of the function with inserted paths, LEOPARD will mistakenly prioritize the inputs that visit these functions while does *not* prioritize the inputs that skip the inserted paths in the function. As a consequence, inputs that execute the inserted path will be more likely to be prioritized.

AFL, Angora, and AFL-Sensitive are also affected by fake paths because fake paths contribute to more code coverage. More generally, any approach that adds more information to edge representation yet still treat edge equally will be affected by anti-fuzzing. Essentially, this is because that edge coverage is treated equally despite the fact that edges have different likelihoods in leading to vulnerabilities.

III. Coverage Accounting

Prior coverage-guided fuzzers [4, 6, 7, 12, 33, 45, 47, 56, 60, 61] are limited as they treat all blocks and edges equally. As a result, these tools may waste time in exploring the codes that are less likely to be vulnerable, and thus are inefficient in finding vulnerabilities. Even worse, prior work can be undermined by current anti-fuzzing techniques [23, 28] which exploit the design deficiency in current coverage measurement.

To mitigate this issue, we propose *coverage accounting*, a new approach to measure edges for input prioritization. Coverage accounting needs to meet two requirements. First, coverage accounting should be lightweighted. One purpose of coverage accounting is to shorten the time to find a vulnerability by prioritizing inputs that are more likely to trigger vulnerabilities. If coverage accounting takes long, it will not be able to shorten the time.

Second, coverage accounting should not rely on taint analysis or symbolic execution. This is because that coverage accounting needs to defend against anti-fuzzing. Since current anti-fuzzing techniques are capable of defeating taint analysis and symbolic execution, we should avoid using these two analyses in coverage accounting.

Based on the intuition that memory corruption vulnerabilities are directly related to memory access operations, we design coverage accounting for memory errors as the measurement of an edge in terms of *future* memory access operations. Furthermore, inspired by HOTracer [27], which treats memory access operations at different levels, we present the latest and future memory access operations from three granularity: *function calls*, *loops*, and *basic blocks*.

Our design is different from known memory access-related measurements. CollAFL [18] counts the total number of memory access operations throughout the execution path, which implies the *history* memory access operations. Wang et al. [53] apply the address rather than the count of memory access. Type-aware fuzzers such as Angora [12], TIFF [26], and ProFuzzer [58] identify inputs that associated to specific memory operations and mutate towards targeted programs or patterns, but they cause higher overhead due to type inference, and that input mutation is separate from input prioritization in our context that could be complementary to our approach.

1) *Function Calls*

On the function call level, we abstract memory access operations as the function itself. Intuitively, if a function was involved in a memory corruption, appearing in the call stack of the crash, then it is likely that the function will be involved again due to patch incompleteness or developers' repeated errors, and we should prioritize the inputs that will visit this function.

TABLE I: Top 20 vulnerability involved functions.

Function	Number	Function	Number
memcpy	80	vsprintf	9
strlen	35	GET_COLOR	7
ReadImage	17	read	7
malloc	15	load_bmp	6
memmove	12	huffcode	6
free	12	strcmp	6
memset	12	new	5
delete	11	getName	5
memcmp	10	strncat	5
getString	9	png_load	5

Inspired by VCCFinder [41], we check the information of disclosed vulnerabilities on the Common Vulnerabilities and Exposures[2] in the latest 4 years to find the vulnerability-involved functions. We crawl the reference webpages on CVE descriptions and the children webpages, extract the call stacks from the reference webpages and synthesize the involved functions. Part of them are shown in Table I (the top 20 vulnerability functions). We observe from this table that the top frequent vulnerability-involved functions are mostly from libraries, especially `libc`, which matches with the general impression that memory operation-related functions in `libc` such as `strlen` and `memcpy` are more likely to be involved in memory corruptions.

Given the vulnerability-involved functions, we assess an edge by the number of vulnerability-involved functions in the destination basic block. Formally, let $\mathcal{F}$ denote for the set of vulnerability-involved functions, let dst_e denote for the destination basic block of edge e, and let $C(b)$ denote for the calling functions in basic block b. For an e, we have:

$$\text{Func}(e) = \mathit{card}\Big(C(dst_e) \cap \mathcal{F}\Big) \quad (1)$$

where $\text{Func}(e)$ represents the metric, and $\mathit{card}(\cdot)$ represents for the cardinality of the variable as a set.

2) Loops

Loops are widely used for accessing data and are closely related to memory errors such as overflow vulnerabilities. Therefore, we introduce the loop metric to incentivize inputs that iterate a loop, and we use the back edge to indicate that. To deal with back edges, we introduce CFG-level instrumentation to track this information, instead of the basic block instrumentation. We construct CFGs for each module of the target program, and analyze the natural loops by detecting back edges [1]. Let function $\text{IsBackEdge}(e)$ be a boolean function outputting whether or not edge e is a back edge. Given an edge indicated by e, we have the loop metric $\text{Loop}(e)$ as follows:

$$\text{Loop}(e) = \begin{cases} 1, \text{if IsBackEdge}(e) = \text{True} \\ 0, \text{otherwise} \end{cases} \quad (2)$$

3) Basic Blocks

The basic block metric abstracts the memory operations that will be executed immediately followed by the edge. As a basic block has only one exit, all instructions will be executed, and the memory access in this basic block will also be enforced. Therefore, it is reasonable to consider the basic block metric as the finest granularity for coverage accounting.

Specifically, we evaluate an edge by the number of instructions that involve memory operations. Let $\text{IsContainMem}(i)$ be a boolean function for whether or not an instruction i contains memory operations. For edge e with destination basic block dst_e, we evaluate the edge by the basic block metric $\text{BB}(e)$ as follows:

$$\text{BB}(e) = \mathit{card}\Big(\big\{i | i \in dst_e \wedge \text{IsContainMem}(i)\big\}\Big) \quad (3)$$

Discussion: Coverage accounting design. One concern is that the choice of the vulnerable is too specific and heuristic-based. We try to make it more general by selecting based on commit history and vulnerability reports, which is also acceptable by related papers [34, 35, 40, 44]. One may argue that this method cannot find vulnerabilities associated with functions that were not involved in any vulnerabilities before or custom functions. This concern is valid, but can be resolved by the other two metrics in a finer granularity as the three coverage accounting metrics are complementary. Moreover, all three metrics contribute to the final results of finding vulnerabilities (More details in subsection “Coverage accounting metrics" in Section VI-D).

Another way to select vulnerable functions is code analysis and there is a recent relevant paper LEOPARD [15] which is according to the complexity score and vulnerability score of functions. The score is calculated based on several code features including loops and memory accesses, which is more like the combination of all three coverage accounting metrics we propose. As the paper of LEOPARD mentioned that it could be used for fuzzing, we set an experiment to compare with it on our data set (More details in subsection “Coverage accounting vs. other metrics" in Section VI-D).

IV. THE DESIGN OF TORTOISEFUZZ

On the high-level, the goal of our design is to prioritize the inputs that are more likely to lead to vulnerable code, and meanwhile ensure the prioritized inputs cover enough code to mitigate the issue that the fuzzer gets trapped or misses vulnerabilities. There are three challenges for the goal. The first challenge is how to properly define the scope of the code to be covered and select a subset of inputs that achieve the complete coverage. Basically AFL's queue culling algorithm guarantees that the selected inputs will cover *all* visited edges. Our insight is that, since memory operations are the prerequisite of memory errors, only security-sensitive edges matter for the vulnerabilities and thus should be fully covered by selected input. Based on this insight, we re-scope the edges from all visited edges to *security-sensitive* only, and we apply AFL's queue culling algorithm on the visited security-sensitive edges. In this way, we are able to select a subset of inputs that cover all visited security-sensitive edges.

The following challenge is how to define security-sensitive with coverage accounting. It is intuitive to set a threshold for the metrics, and then define edges exceeding the threshold as security sensitive. We set the threshold conservatively: edges

[2]For prototype, we use CVE dataset, https://cve.mitre.org/

are security sensitive as long as the value of the metrics is above 0. We leave the investigation on the threshold as future work (see Section VII).

The last challenge is how to make fuzzing evolve towards vulnerabilities. Our intuition is that, the more an input hits a security-sensitive edge, the more likely the input will evolve to trigger a vulnerability. We prioritize an input with hit count, based on the proposed metrics for coverage accounting.

Based on the above considerations, we decide to design TortoiseFuzz based upon AFL, and remain the combination of input filtering and queue culling for input prioritization. TortoiseFuzz, as a greybox coverage-guided fuzzer with coverage accounting for input prioritization, is lightweighted and robust to anti-fuzzing. For the ease of demonstration, we show the algorithm of AFL in Algorithm 1, and explain our design (marked in grey) based on AFL's algorithm.

Algorithm 1 Fuzzing algorithm with coverage accounting

```
1:  function FUZZING(Program, Seeds)
2:      P ← INSTRUMENT(Program, CovFb, AccountingFb)          ▷ Instr. Phase
3:      // AccountingFb is FunCallMap, LoopMap, or InsMap

4:      INITIALIZE(Queue, CrashSet, Seeds)
5:      INITIALIZE(CovFb, accCov, TopCov)
6:      INITIALIZE(AccountingFb, accAccounting, TopAccounting)
7:      // accAccounting is MaxFunCallMap, MaxLoopMap, or MaxInsMap
8:      repeat                                                ▷ Fuzzing Loop Phase
9:          input ← NEXTSEED(Queue)
10:         NumChildren ← MUTATEENERGY(input)
11:         for i = 0 → NumChildren do
12:             child ← MUTATE(input)
13:             IsCrash, CovFb, AccountingFb ← RUN(P, child)
14:             if IsCrash then
15:                 CrashSet ← CrashSet ∪ child
16:             else if SAVE_IF_INTERESTING(CovFb, accCov) then
17:                 TopCov, TopAccounting ←
18:                     UPDATE(child, CovFb, AccountingFb, accAccounting)
19:                 Queue ← Queue ∪ child
20:             end if
21:         end for
22:         CULL_QUEUE(Queue, TopCov, TopAccounting)
23:     until time out
24: end function
```

A. Framework

The process of TortoiseFuzz is shown in Algorithm 1. TortoiseFuzz consists of two phases: instrumentation phase and fuzzing loop phase. In the instrumentation phase (Section IV-B), the target program is instrumented with codes for preliminary analysis and runtime execution feedback. In the fuzzing loop phase (Section IV-C), TortoiseFuzz iteratively executes the target program with testcases, appends interesting samples to the fuzzing queue based on the execution feedback, and selects inputs for future iterations.

B. Instrumentation Phase

The instrumentation phase is to insert runtime analysis code into the program. For source code, we add the analysis code during compilation; otherwise, we rewrite the code to insert the instrumentation. If the target requires specific types of inputs, we modify the I/O interface with instrumentation. The inserted runtime analysis code collects the statistics for coverage and security sensitivity evaluation.

C. Fuzzing Loop Phase

The fuzzing loop is described from line 8 to 23 in Algorithm 1. Before the loop starts, TortoiseFuzz first creates a sample queue $Queue$ from the initial seeds and a set of crashes $CrashSet$ (line 4). The execution feedback for each sample is recorded in the coverage feedback map (i.e., $CovFb$ at line 5) and accounting feedback map (i.e., $AccountingFb$ at line 6). The corresponding maps $accCov$ (line 5) and $accAccounting$ (line 6) are global accumulated structures to hold all covered transitions and their maximum hit counts. The $TopCov$ and $TopAccounting$ are used to prioritize samples.

For each mutated sample, TortoiseFuzz feeds it to the target program and reports if the return status is crashed. Otherwise, it uses the function Save_If_Interesting to append it to the sample queue $Queue$ if it matches the input filter conditions (new edges or hit bucket change) (line 16). It will also update the structure $accCov$.

For the samples in $Queue$, the function NextSeed selects a seed for the next test round according to the probability (line 9), which is determined by the *favor* attribute of the sample. If the value of *favor* is 1, then the probability is 100%; otherwise it is 1%. The origin purpose of *favor* is to have a minimal set of samples that could cover all edges seen so far, and turn to fuzz them at the expense of the rest. We improve the mechanism to prioritize mutated samples with two steps, *Update* (line 18) and *Cull_Queue* (line 22). More specifically, *Update* will update the structure *accAccounting* and return the top rated lists *TopCov* and *TopAccounting*, which are used in the following step of function *Cull_Queue*.

1) Updating Top Rated Candidates

To prioritize the saved interesting mutations, greybox fuzzers (e.g., AFL) maintain a list of entries *TopCov* for each edge $edge_i$ to record the best candidates, $sample_j$, that are more favorable to explore. As shown in Formula 4, $sample_j$ is "favor" for $edge_i$ as the sample can cover $edge_i$ and there are no previous candidates, or if it has less cost than the previous ones (i.e., *execution latency multiplied by file size*).

$$TopCov[edge_i] = \begin{cases} sample_j, & CovFb_j[edge_i] > 0 \\ & \quad \wedge\ (TopCov[edge_i] = \emptyset \\ & \quad \vee\ \text{IsMin}(exec_time_j * size_j)) \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

Cost-favor entries are not sufficient for the fuzzer to keep the sensitivity information to the memory operations; hence, TortoiseFuzz maintains a list of entries for each memory-related edge to record the "memory operation favor". As Formula 5 shows, if there is no candidate for $edge_i$, or if the $sample_j$ could max the hit count of edge $edge_i$, we mark it as "favor". If the hit count is the same with the previous saved one, we mark it as "favor" if the cost is less. The *AccountingFb*

and *accAccounting* are determined by coverage accounting.

$$TopAccounting[edge_i] = \begin{cases} sample_j, & (TopAccounting[edge_i] == \emptyset \land \ CovFb_j[edge_i] > 0) \\ & \quad \lor \ AccountingFb_j[edge_i] > accAccounting[edge_i] \\ & \quad \lor \ (AccountingFb_j[edge_i] == accAccounting[edge_i] \\ & \quad \land \text{IsMin}(exec_time_j * size_j)) \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

2) *Queue Culling*

The top-rated candidates recorded by *TopAccounting* are a superset of samples that can cover all the security-sensitive edges seen so far. To optimize the fuzzing effort, as shown in Algorithm 2, TortoiseFuzz re-evaluates all top rated candidates after each round of testing to select a quasi-minimal subset of samples that cover all of the accumulated memory related edges and have not been fuzzed. First, we create a temporal structure *Temp_map* to hold all the edges seen up to now. During traversing the seed queue *Queue*, if a sample is labeled with “favor”, we will choose it as final “favor” (line 9). Then the edges covered by this sample is computed and the temporal *Temp_map* (line 10) is updated. The process proceeds until all the edges seen so far are covered. With this algorithm, we select favorable seeds for the next generation and we expect they are more dangerous to memory errors (line 13).

However, TortoiseFuzz prefers to exploring program states with less breadth than the original path coverage. This may result in a slow increase of samples in *Queue* and fewer samples with the *favor* attributes. To solve this problem, TortoiseFuzz uses the original coverage-sensitive top rated entries *TopCov* to re-cull the *Queue* while there is no favor sample in the *Queue* (line 15-24). Also, whenever the *TopAccounting* is changed (line 5), TortoiseFuzz will switch back to the security-sensitive strategy.

Algorithm 2 Cull Queue

```
 1: function CULL_QUEUE(Queue, TopCov, TopAccounting)
 2:     for q = Queue.head → Queue.end do
 3:         q.favor = 0
 4:     end for
 5:     if IsChanged(TopAccounting) then
 6:         Temp_map ← accCov[MapSize]
 7:         for i = 0 → MapSize do
 8:             if TopAccounting[i] && TopAccounting[i].unfuzzed then
 9:                 TopAccounting[i].favor = 1
10:                 UPDATE_MAP(TopAccounting[i], Temp_map)
11:             end if
12:         end for
13:         SYN(Queue, TopAccounting)
14:     else
15:         // switch back to TopCov with coverage-favor
16:         for i = 0 → MapSize do
17:             Temp_map ← accCov[MapSize]
18:             if TopCov[i] && Temp_map[i] then
19:                 TopCov[i].favor = 1
20:                 UPDATE_MAP(TopCov[i], Temp_map)
21:             end if
22:         end for
23:         SYN(Queue, TopCov)
24:     end if
25: end function
```

Discussion: Defending against anti-fuzzing. Current anti-fuzzing techniques defeat prior fuzzing tools by inserting fake paths that trap fuzzers, adding a delay in error-handling code, and obfuscating code to slow down taint analysis and symbolic execution. TortoiseFuzz, along with coverage accounting, is robust to code obfuscation as it does not require taint analysis or symbolic execution. It is also not highly affected by anti-fuzzing because input prioritization helps to avoid the execution of error-handling code since error-handling code do not typically contain intensive memory operation. Also coverage accounting is robust to inserted fake branches created by Fuzzification [28], which the branches are composed of `pop` and `ret`. As Coverage accounting does not consider `pop` and `ret` as security-sensitive operations, it will not prioritize inputs that visit fake branches.

One may argue that a simple update for anti-fuzzing will defeat TortoiseFuzz, such as adding memory operations in fake branches. However, since memory access costs much more than other operations such as arithmetic operations, adding memory access operations in fake branches may cause slowdown and affect the performance for normal inputs, which is not acceptable for real-world software. Therefore, one has to carefully design fake branches that defeat TortoiseFuzz and keep reasonable performance, which is much harder than the current anti-fuzzing methods. Therefore, we argue that although TortoiseFuzz is not guaranteed to defend against all anti-fuzzing techniques now and future, it will significantly increase the difficulty of successful anti-fuzzing.

V. Implementation

TortoiseFuzz is implemented based on AFL [60]. Besides the AFL original implementation, TortoiseFuzz consists of about 1400 lines of code including instrumentation (∼700 lines in C++) and fuzzing loop (∼700 lines in C). We also wrote a python program of 34 lines to crawl the vulnerability reports. For the function call level coverage accounting, we get the function names in instructions by calling *getCalledFunction()* in the LLVM pass, and calculate the weight value by matching with the list of high-risk functions in Table I. For the loop level coverage accounting, we construct the CFG with adjacency matrix and then use the depth-first search algorithm to traverse the CFG and mark the back edges. For the basic block level coverage accounting, we mark the memory access characteristics of the instructions with the *mayReadFromMemory()* and *mayWriteToMemory()* functions from LLVM [30].

VI. Evaluation

In this section, we evaluate coverage accounting by testing TortoiseFuzz on real-world applications. We will answer the following research questions:

- **RQ1**: Is TortoiseFuzz able to find real-world zero-day vulnerabilities?
- **RQ2**: How do the results of TortoiseFuzz compare to previous greybox or hybrid fuzzers in real-world programs?
- **RQ3**: How do the results of the three coverage accounting metrics compare to other coverage metrics or input prioritization approaches?
- **RQ4**: Is coverage accounting able to cooperate with other fuzzing techniques and help improve vulnerability discovery?
- **RQ5**: Is coverage accounting robust against the current anti-fuzzing technique?

TABLE II: Compared fuzzers.

Fuzzer	Year	Type	Open	Target	Select
AFL	2016	greybox	Y	S/B[1]	✓
AFLFast	2016	greybox	Y	S/B	✓
Steelix	2017	greybox	N	S	
VUzzer	2017	hybrid	Y[2]	B	
CollAFL	2018	greybox	N	S	
FairFuzz	2018	greybox	Y	S	✓
T-fuzz	2018	hybrid	Y[3]	B	
QSYM	2018	hybrid	Y	S	✓
Angora	2018	hybrid	Y	S	✓
MOPT	2019	greybox	Y	S	✓
DigFuzz	2019	hybrid	N	S	
ProFuzzer	2019	hybrid	N	S	

[1] S: target source code, B: target binary.
[2] VUzzer depends on external IDA Pro and pintool, and instrumenting with real-world program is too expensive to be scalable in our experiment environment. Also, VUzzer did not perform as well as other hybrid tools such as QSYM. Under such condition, we did not include VUzzer in real-world program experiment.
[3] Some components of these tools cannot work for all binaries.

A. *Experiment Setup*

Dataset. We collected 30 applications from the papers published from 2016 to 2019. The applications include image parsing and processing libraries, text parsing tools, an assembly tool, multimedia file processing libraries, language translation tools, and so on. We selected the latest version of the testing applications by the time we ran the experiment.

One may argue that the experiment dataset is lack of other test suites such as LAVA-M. We find that LAVA-M does not fit for evaluating fuzzer effectiveness since it does not reflect the real-world scenarios. We will further discuss the choice of dataset in Section VII.

In our evaluation, we found that there are 18 applications from which no fuzzers found any vulnerabilities. For ease of demonstration, we will only show the results for the 12 applications with found vulnerabilities throughout the rest of the evaluation.

Compared fuzzers. We collected recent fuzzers published from 2016 to 2019 as the candidate for comparison, as shown in Table II. We consider each fuzzer regarding whether or not it is open-source, and whether or not it can run on our experiment dataset. We filtered tools that are not open-source or do not scale to our test real-world programs and thus cannot be compared. The remained compared fuzzers are 4 greybox fuzzers (AFL, AFLFast, FairFuzz and MOPT) and 2 hybrid fuzzers (QSYM and Angora). For detailed explanations for each fuzzer, we refer readers to the footnote in Table II.

Environment and process. We ran our experiments on eight identical servers with 32 CPU Intel(R) Xeon(R) CPU E5-2630 V3@2.40GHZ cores, 64GB RAM, and 64-bits Ubuntu 16.04.3 TLS. For each target application, we configured all fuzzers with the same seed[3] and dictionary set[4]. We ran each fuzzer with each target program for 140 hours according to CollAFL [18], and we repeated all experiments for 10 times as advised by Klees et al. [29].

[3]We select an initial seed randomly from the testcase set provided by the target program.

[4]We do not use the dictionary in the experiment.

We identified vulnerabilities in two steps. Given reported crashes, we first filtered out invalid and redundant crashes using a self-written script with ASan [46] then manually inspected the remaining crashes and reported those that are security related. For code coverage, we used gcov [20], which is a well-known coverage analysis tool.

B. *RQ1: Finding Zero-day Vulnerabilities*

Table III shows the union of the real-world vulnerabilities identified by TortoiseFuzz in the 10 times of the experiment. The table presents each discovered vulnerability with its ID, the corresponding target program, the vulnerability type and whether it is a zero-day vulnerability. In total, TortoiseFuzz found 56 vulnerabilities in 10 different types, including stack buffer overflow, heap buffer overflow, use after free, and double free vulnerabilities, many of which are critical and may lead to severe consequences such as arbitrary code execution. Among the 56 found vulnerabilities, 20 vulnerabilities are zero-day vulnerabilities, and 15 vulnerabilities have been confirmed with a CVE identification[5]. The result indicates that TortoiseFuzz is able to identify a reasonable number of zero-day vulnerabilities from real-world applications.

C. *RQ2: TortoiseFuzz vs. Other Fuzzers in Real-world Applications*

In this experiment, we tested and compared TortoiseFuzz with greybox fuzzers and hybrid fuzzers on real-world applications. We evaluated each fuzzer by three metrics: discovered vulnerabilities, code coverage, and performance.

Discovered vulnerabilities. Table IV shows the average and the maximum numbers of vulnerabilities found by each fuzzer among 10 repeated runs. The table also shows the p-value of the Mann-Whitney U test between TortoiseFuzz and a comparing fuzzer regarding the total number of vulnerabilities found from all target programs in each of the 10 repeated experiment.

Our experiment shows that TortoiseFuzz is more effective than the other testing greybox fuzzers in finding vulnerabilities from real-world applications. TortoiseFuzz detected 41.7 vulnerabilities on average and 50 vulnerabilities in maximum, which outperformed all the other greybox fuzzers. Comparing to FairFuzz, the second best-performed greybox fuzzers, TortoiseFuzz found 43.9% more vulnerabilities on average and 31.6% more in maximum. Additionally, we compared the set of vulnerabilities in the best run of each fuzzer, and we observed that TortoiseFuzz covered all but 1 vulnerability found by the other fuzzer, and TortoiseFuzz found 10 more vulnerabilities that none of the other fuzzers discovered.

For hybrid fuzzers, TortoiseFuzz showed a better result than Angora and a comparable result with QSYM. TortoiseFuzz outperformed Angora by 61.6% on the sum of average and by 56.3% on the sum of the maximum numbers of vulnerabilities for each target programs among the 10 times of the experiment. TortoiseFuzz also detected more or equal vulnerabilities from 91.7% (11/12) of the target applications. TortoiseFuzz found slightly more vulnerabilities than QSYM

[5]CVE-2018-17229 and CVE-2018-17230 are contained in both exiv2 and new_exiv2.

TABLE III: Real-world Vulnerabilities found by TortoiseFuzz.

Program	Version	ID	Vulnerability Type	New
exiv2	0.26	CVE-2018-16336	heap-buffer-overflow	✓
		CVE-2018-17229	heap-buffer-overflow	✓
		CVE-2018-17230	heap-buffer-overflow	✓
		issue_400	heap-buffer-overflow	✓
		issue_460	stack-buffer-overflow	-
		CVE-2017-11336	heap-buffer-overflow	-
		CVE-2017-11337	invalid free	-
		CVE-2017-11339	heap-buffer-overflow	-
		CVE-2017-14857	invalid free	-
		CVE-2017-14858	heap-buffer-overflow	-
		CVE-2017-14861	stack-buffer-overflow	-
		CVE-2017-14865	heap-buffer-overflow	-
		CVE-2017-14866	heap-buffer-overflow	-
		CVE-2017-17669	heap-buffer-overflow	-
		issue_170	heap-buffer-overflow	-
		CVE-2018-10999	heap-buffer-overflow	-
new_exiv2	0.26	CVE-2018-17229	heap-buffer-overflow	✓
		CVE-2018-17230	heap-buffer-overflow	✓
		CVE-2017-14865	heap-buffer-overflow	-
		CVE-2017-14866	heap-buffer-overflow	-
		CVE-2017-14858	heap-buffer-overflow	-
exiv2_9.17	0.26	CVE-2018-17282	null pointer dereference	✓
nasm	2.14rc4	CVE-2018-8882	stack-buffer-under-read	-
		CVE-2018-8883	stack-buffer-over-read	-
		CVE-2018-16517	null pointer dereference	-
		CVE-2018-19209	null pointer dereference	-
		CVE-2018-19213	memory leaks	-
gpac	0.7.1	CVE-2019-20165	null pointer dereference	✓
		CVE-2019-20169	heap-use-after-free	✓
		CVE-2018-21017	memory leaks	✓
		CVE-2018-21015	Segment Fault	✓
		CVE-2018-21016	heap-buffer-overflow	✓
		issue_1340	heap-use-after-free	-
		issue_1264	heap-buffer-overflow	-
		CVE-2018-13005	heap-buffer-over-read	-
		issue_1077	heap-use-after-free	-
		issue_1090	double-free	-
libtiff	4.0.9	CVE-2018-15209	heap-buffer-overflow	✓
		CVE-2018-16335	heap-buffer-over-read	✓
liblouis	3.7.0	CVE-2018-11440	stack-buffer-overflow	-
		issue_315	memory leaks	-
ngiflib	0.4	issue_10	stack-buffer-overflow	✓
		CVE-2019-16346	heap-buffer-overflow	✓
		CVE-2019-16347	heap-buffer-overflow	✓
		CVE-2018-11575	stack-buffer-overflow	-
		CVE-2018-11576	heap-buffer-over-read	-
libming	0_4_8	CVE-2018-13066	memory leaks	-
		(2 similar crashes)	memory leaks	-
catdoc	0_95	crash	memory leaks	-
		crash	Segment Fault	-
		CVE-2017-11110	heap-buffer-underflow	-
tcpreplay	4.3	CVE-2018-20552	heap-buffer-overflow	✓
		CVE-2018-20553	heap-buffer-overflow	✓
flvmeta	1.2.1	issue_13	null pointer dereference	✓
		issue_12	heap-buffer-overflow	-

on average and equal vulnerabilities in maximum. For each target program, specifically, TortoiseFuzz performed better than QSYM in 7 programs, equally in 2 programs, and worse in 3 programs. Based on the Mann-Whitney U test, the result between TortoiseFuzz and QSYM was not statistically significant, and thus we consider TortoiseFuzz comparable to QSYM in finding vulnerabilities from real-world programs.

Furthermore, we compared the union set of the discovered vulnerabilities across 10 repeated runs between TortoiseFuzz and QSYM. While TortoiseFuzz missed 9 vulnerabilities found by QSYM, only one of them is a zero-day vulnerability. The zero-day vulnerability is in the `parse_mref` function of nasm. We analyzed the missing cases and found that, some of the vulnerabilities are protected by conditional branches related to the input file, which does not belong to any of our metrics and thus the associated inputs cannot be prioritized.

Overall, our experiment results show that TortoiseFuzz outperformed AFL, AFLFast, FairFuzz, MOPT, and Angora, and it is comparable to QSYM in finding vulnerabilities.

Code coverage. In addition to the number of found vulnerabilities, we measured the code coverage of the testing fuzzers across different target programs. Although coverage accounting does not aim to improve code coverage, measuring code coverage is still meaningful, as it is an import metrics for evaluating program testing techniques. We also want to investigate if coverage accounting affects code coverage in the fuzzing process.

To investigate the impact on code coverage caused by coverage accounting, we compare the code coverage between TortoiseFuzz and AFL, the fuzzer based on which coverage accounting is implemented. Table V shows the average code coverage of all fuzzers performing on the target programs, and Table VI shows the p-value of the Mann-Whitney U test between TortoiseFuzz and other fuzzers. Based on Table V, we observe that TortoiseFuzz had a better coverage than AFL on average for 75% (9/12) of the target programs. In terms of statistical significance, 3 out of 12 programs are statistically different in coverage, and TortoiseFuzz has a higher average value for all the three cases, which implies that TortoiseFuzz is statistically better than AFL in code coverage. Therefore, coverage accounting does not affect code coverage in fuzzing process.

Comparing TortoiseFuzz to other fuzzers, we observe that although TortoiseFuzz does not aim to high coverage, its performance is fair among all fuzzers. Most of the results are not statistically significant between TortoiseFuzz and AFL, AFLFast, and FairFuzz. Between TortoiseFuzz and MOPT, TortoiseFuzz performed statistically better in three cases, whereas MOPT performed statistically better in two cases. TortoiseFuzz's results are statistically higher than that of Angora in most cases, not as good as that of QSYM.

Furthermore, we study the variability of the code coverage of each testing fuzzer and target program, shown in Figure 2 and Table VII. We observe from the figure that the performance of TortoiseFuzz in code coverage is stable in most of the testing cases: the interquartile range is below 2% for 11 out of 12 of the testing programs.

Performance. Given that TortoiseFuzz had a comparable result with QSYM, we compare the resource performance of TortoiseFuzz to that of the hybrid fuzzer QSYM. For each of the 10 repeated experiment, we logged the memory usage of QSYM and TortoiseFuzz every five seconds, and we show the memory usage of each fuzzer and each targeting program in Figure 3. The figure indicates that TortoiseFuzz spent less memory resources than QSYM, which reflects the fact that hybrid fuzzers need more resources to execute heavy-weighted analyses such as taint analysis, concolic execution, and constraint solving.

Case study. To better understand the internal of why Tortoise-

TABLE IV: The Average Number of Vulnerabilities Identified by Each Fuzzer.

Program	Grey-box Fuzzers										Hybrid Fuzzers			
	TortoiseFuzz		AFL		AFLFast		FairFuzz		MOPT		Angora		QSYM	
	Average	Max	Average	Max	Average	Max	Average	Max	Average	Max	Average	Max	Average	Max
exiv2	9.7	12	9.0	12	5.4	9	7.1	10	8.7	11	10.0	13	8.3	10
new_exiv2	5.0	5	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	6.5	9
exiv2_9.17	1.0	1	0.9	1	0.4	1	0.7	1	0.8	1	0.0	0	1.8	2
gpac	7.6	9	3.5	6	4.8	7	6.6	9	4.0	6	6.0	8	7.5	9
liblouis	1.2	2	0.3	1	0.0	0	0.9	2	0.1	1	0.0	0	2.3	3
libming	3.0	3	2.9	3	3.0	3	3.0	3	0.0	0	3.0	3	3.0	3
libtiff	1.2	2	0.0	0	0.0	0	0.0	0	0.0	0	0.0	0	0.4	1
nasm	3.0	4	1.7	2	2.2	3	2.2	3	3.8	5	1.8	2	2.8	4
ngiflib	4.7	5	4.4	5	3.2	5	4.0	5	2.7	5	3.0	4	4.5	5
flvmeta	2.0	2	2.0	2	2.0	2	2.0	2	2.0	2	2.0	2	2.0	2
tcpreplay	1.2	2	0.0	0	0.0	0	0.5	1	0.0	0	-*	-*	0.0	0
catdoc	2.1	3	1.3	2	1.2	2	2.0	2	0.3	1	0.0	0	2.0	2
SUM	41.7	50	26.0	34	22.2	32	29.0	38	22.4	32	25.8	32	41.1	50
p-value of the Mann-Whitney U test			0.0001668		0.0001668		0.0001649		0.0001659		0.0001668		1.0	

* Angora run abnormally. For all 360 pcap files included in the test suit, Angora reported the error log "There is none constraint in the seeds".

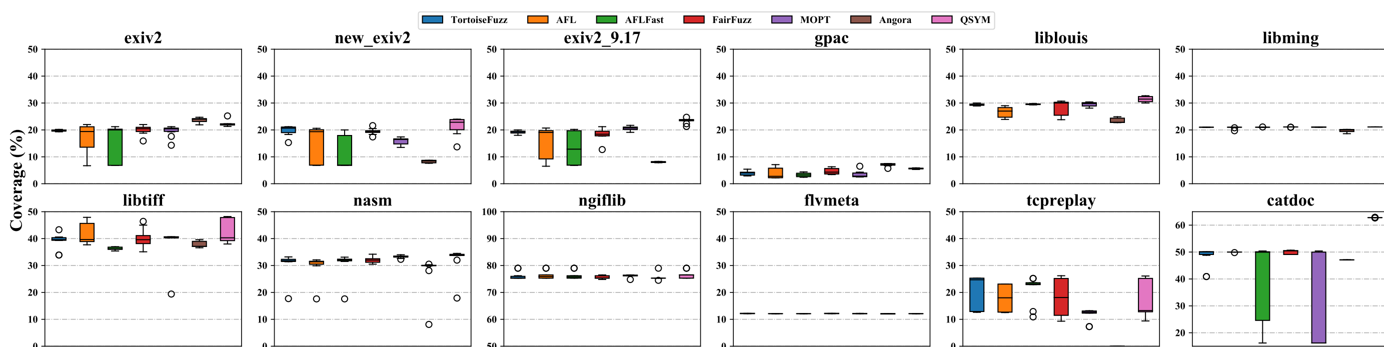

Fig. 2: The variability of the code coverage of different fuzzers with different target programs.

TABLE V: Code coverage (average in 10 runs) of the target applications achieved by various fuzzers. The highest values among fuzzers are highlighted in blue.

Program	Grey-box Fuzzers					Hybrid Fuzzers	
	TortoiseFuzz	AFL	AFLFast	FairFuzz	MOPT	Angora	QSYM
exiv2	19.77%	16.65%	15.00%	20.02%	19.53%	**23.65%**	22.23%
new_exiv2	19.83%	14.71%	11.67%	19.44%	15.86%	8.29%	**21.40%**
exiv2_9.17	19.16%	15.41%	13.27%	18.37%	20.53%	8.03%	**23.38%**
gpac	3.92%	3.86%	3.31%	4.64%	3.47%	**7.07%**	5.63%
liblouis	29.44%	26.65%	29.53%	28.30%	29.47%	23.41%	**31.42%**
libming	21.00%	20.85%	21.01%	21.08%	21.05%	19.66%	**21.10%**
libtiff	39.00%	41.62%	36.27%	40.09%	38.42%	37.75%	**42.77%**
nasm	30.64%	29.83%	30.71%	31.98%	**33.27%**	27.70%	32.22%
ngiflib	76.13%	76.40%	76.31%	75.67%	76.15%	75.59%	**76.52%**
flvmeta	**12.16%**	12.10%	12.10%	**12.16%**	12.13%	12.05%	12.10%
tcpreplay	20.17%	17.89%	**21.34%**	18.20%	11.71%	-	17.61%
catdoc	48.12%	49.98%	39.90%	49.98%	29.74%	47.10%	**62.80%**

TABLE VI: p-values of the Mann-Whitney U test on the code coverage in 10 runs.

Program	AFL	AFLFast	FairFuzz	MOPT	Angora	QSYM
exiv2	6.23E-01	7.04E-01	1.40E-01	2.25E-01	1.80E-04	1.79E-04
exiv2_new	2.30E-02	1.65E-03	2.40E-01	1.31E-03	1.57E-04	1.30E-01
exiv2_9.17	6.23E-01	3.62E-01	2.56E-01	4.56E-03	1.71E-04	1.78E-04
gpac	4.48E-01	1.72E-01	2.89E-01	9.54E-02	1.75E-04	2.76E-04
liblouis	2.75E-04	6.15E-01	3.23E-01	5.69E-01	1.54E-04	2.03E-04
libming	1.68E-01	3.68E-01	4.41E-04	1.37E-02	5.94E-05	1.59E-05
libtiff	7.61E-01	2.49E-02	7.90E-01	9.90E-02	1.10E-01	1.83E-01
nasm	1.01E-01	6.21E-01	8.50E-01	1.63E-03	2.07E-03	5.61E-03
ngiflib	4.39E-01	5.18E-01	9.32E-01	1.43E-01	2.13E-01	2.60E-01
flvmeta	5.02E-03	5.02E-03	1.00E+00	2.04E-01	1.35E-03	5.02E-03
tcpreplay	9.29E-02	4.24E-01	4.02E-01	2.02E-02	-	6.75E-01
catdoc	2.18E-01	6.56E-01	9.67E-02	7.46E-02	1.55E-02	1.19E-04

Fuzz managed to find zero-days vulnerabilities, we conduct a case study to investigate the fuzzing process and compare TortoiseFuzz to other fuzzers such as AFL. Figure 4 shows the fuzzing process of TortoiseFuzz and AFL for finding CVE-2018-16335[6]. The *Seed ID* indicates the ID of the testing inputs generated by each fuzzer. The lines in the figure shows the evolution of the generated seeds along with the fuzzing loop. The label of the nodes in the line of TortoiseFuzz shows the metric causing the seed to be prioritized.

Based on the figure, we find that TortoiseFuzz and AFL and TortoiseFuzz deviated in the second round of fuzzing. AFL prioritized other seeds over Seed 147, since Seed 147 is memory-intensive and results in longer execution time. However, we consider memory operations as a coverage accounting metric, and thus TortoiseFuzz prioritized the seed. As a result, Seed 147 evolved and finally became the input that triggers the vulnerability. This case indicates that memory operations, although cost longer execution time, helps to generate inputs that triggering memory corruption errors and thus should be

[6]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16335

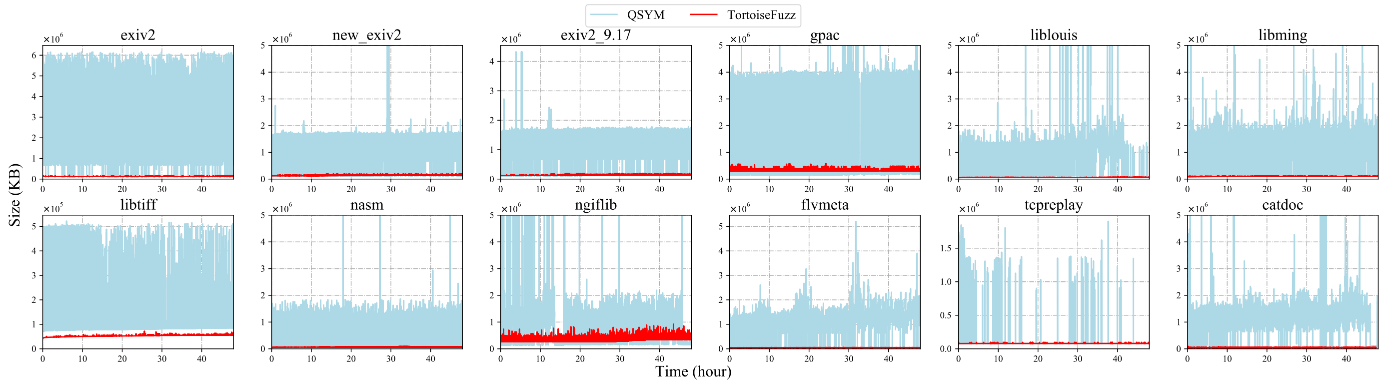

Fig. 3: Memory resource usage of QSYM and TortoiseFuzz.

TABLE VII: The variance of the code coverage.

Program	Grey-box Fuzzers					Hybrid Fuzzers	
	TortoiseFuzz	AFL	AFLFast	Fairfuzz	MOPT	Angora	QSYM
exiv2	7.61E-06	3.18E-03	4.47E-03	2.71E-04	4.00E-04	6.52E-05	1.08E-04
exiv2_new	3.29E-04	4.11E-03	3.53E-03	1.16E-04	1.71E-04	2.27E-05	9.88E-04
exiv2_9.17	3.20E-05	3.32E-03	4.10E-03	4.99E-04	6.90E-05	3.61E-06	8.00E-05
gpac	5.58E-05	3.51E-04	5.15E-05	1.18E-04	1.43E-04	2.78E-05	3.21E-06
liblouis	1.10E-05	3.55E-04	3.01E-06	8.41E-04	6.52E-05	9.53E-05	1.10E-04
libming	7.70E-34	1.51E-05	9.00E-08	1.60E-07	2.50E-07	3.60E-05	7.70E-34
libtiff	7.72E-04	1.63E-03	2.94E-05	1.10E-03	4.02E-03	1.50E-04	1.86E-03
nasm	1.90E-03	1.70E-03	1.93E-03	1.12E-04	2.28E-05	4.31E-03	2.32E-03
ngiflib	2.13E-04	1.96E-04	1.96E-04	3.38E-05	2.21E-05	1.35E-04	1.83E-04
flvmeta	2.40E-07	0.00E+00	0.00E+00	2.40E-07	2.10E-07	2.50E-07	0.00E+00
tcpreplay	3.65E-03	2.68E-03	2.31E-03	4.74E-03	4.98E-04	-	4.21E-03
catdoc	1.32E-03	3.60E-07	2.41E-02	5.22E-05	2.75E-02	0.00E+00	4.00E-07

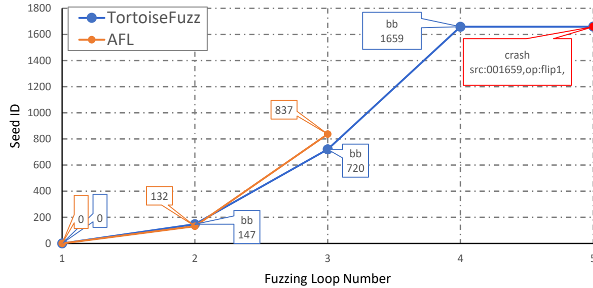

Fig. 4: Test case generation process of CVE-2018-16335

taken as a metric for input prioritization.

D. RQ3: Coverage Metrics

Recall that we proposed three metrics for coverage accounting: Function calls, loops, and basic blocks. In this subsection, we evaluate the effectiveness of each metric, and we compare our combination of the metrics with other coverage-related metrics and input prioritization approach. We ran the three metrics separately for 140 hours and repeated for 10 times, and we represent the result based on these separate runs.

Internal investigation of the three metrics of coverage accounting. In this experiment, we investigate the effectiveness of each metric and their individual contribution to the overall metrics. We ran the three metrics separately for 140 hours and repeated for 10 times, and we represent the result based on these separate runs. Table VIII shows the code coverage and the number of discovered vulnerabilities of each coverage accounting metric. In the table, *func* represents the function call metric, *loop* represents the loop metric, and *bb* represents the basic block metric. Running with the real-world programs, the *loop*, *func*, and *bb* metrics found 28.2, 24.6, and 28.3 vulnerabilities on average, respectively. All metrics found a non-negligible number of vulnerabilities exclusively, as shown in Figure 5. This implies that the three metrics complement each other and are all necessary for coverage accounting.

TABLE VIII: Code coverage and vulnerabilities of the three strategies of TortoiseFuzz.

Program	Code coverage			Vulnerabilities					
				Average			Max		
	func	bb	loop	func	bb	loop	func	bb	loop
exiv2	17.39%	14.64%	17.56%	6.0	5.7	6.6	10	8	12
new_exiv2	18.66%	19.20%	19.71%	3.0	0.0	2.5	5	0	5
exiv2_9.17	16.75%	15.70%	14.80%	0.4	0.8	0.3	1	1	1
gpac	3.73%	3.77%	3.77%	4.6	4.2	4.3	6	7	9
liblouis	26.92%	25.64%	24.11%	1.0	0.5	0.7	1	1	2
libming	20.62%	20.72%	20.02%	3.0	3.0	3.0	3	3	3
libtiff	37.37%	38.79%	37.33%	0.0	0.8	0.3	0	2	1
nasm	29.07%	29.48%	28.80%	1.9	2.5	2.3	3	3	3
ngiflib	76.04%	76.04%	76.04%	3.9	3.6	3.8	5	5	5
flvmeta	12.10%	12.10%	12.10%	2.0	2.0	2.0	2	2	2
tcpreplay	17.24%	16.98%	17.56%	0.4	0.0	0.8	1	0	2
catdoc	48.07%	48.04%	47.98%	2.0	1.5	1.7	2	2	3

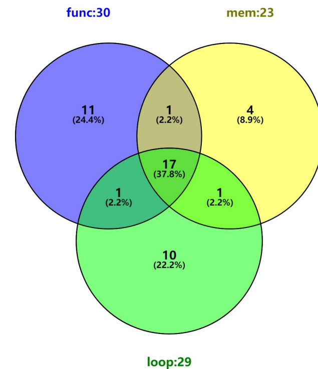

Fig. 5: The set of the bugs found by the 3 coverage accounting metrics from real-world programs (in the best run).

TABLE IX: The number of found vulnerabilities associated with the metrics of coverage accounting and AFL-Sensitive.

Program	Vulnerabilities											
	TortoiseFuzz				AFL-Sensitive							
	func	bb	loop	ALL	bc	ct	ma	mw	n2	n4	n8	ALL
objdump	0	1	0	1	0	0	0	0	1	1	0	1
readelf	0	0	0	0	0	0	0	0	0	0	0	0
strings	0	0	0	0	0	0	0	0	0	0	0	0
nm	0	1	0	1	0	1	0	1	0	0	1	1
size	1	1	1	1	0	1	0	0	1	1	1	1
file	0	0	0	0	0	0	0	0	0	0	0	0
gzip	0	0	0	0	0	0	0	0	0	0	0	0
tiffset	0	1	1	1	0	0	0	0	0	0	0	0
tiff2pdf	1	0	0	1	0	0	0	0	0	0	0	0
gif2png	3	5	5	5	4	4	5	4	4	4	5	5
info2cap	7	5	10	10	9	7	5	5	10	7	7	10
jhead	0	0	0	0	0	0	0	0	0	0	0	0
SUM	12	14	17	20	13	13	10	10	16	13	14	18

TABLE X: The code coverage associated with the metrics of coverage accounting and AFL-Sensitive. The highest values are highlighted in blue.

Program	Code coverage(%)									
	TortoiseFuzz			AFL-Sensitive						
	func	bb	loop	bc	ct	ma	mw	n2	n4	n8
objdump	6.30	**9.00**	8.50	7.80	5.60	6.00	5.70	7.80	7.90	6.90
readelf	21.50	35.60	**37.40**	34.70	33.10	25.70	28.90	33.00	33.30	35.00
strings	**0.20**	**0.20**	**0.20**	**0.20**	**0.20**	**0.20**	**0.20**	**0.20**	**0.20**	**0.20**
nm	4.60	**12.20**	11.20	10.80	9.50	5.40	5.20	6.30	10.50	9.90
size	5.70	5.70	5.60	**6.30**	5.40	4.70	4.50	6.10	6.20	5.50
file	34.20	**34.50**	34.30	22.20	22.90	23.30	23.40	22.80	24.60	22.90
gzip	38.60	37.70	37.70	38.20	38.60	**38.90**	36.20	38.60	38.60	38.70
tiffset	30.10	30.40	**30.50**	10.00	10.00	10.50	10.00	10.00	10.00	10.00
tiff2pdf	**40.70**	40.50	38.70	31.30	33.40	33.10	29.40	30.80	33.40	31.60
gif2png	**73.60**	73.20	73.20	72.80	72.80	69.10	64.40	72.70	73.40	73.40
info2cap	41.20	40.90	**42.10**	40.70	41.00	34.30	35.60	41.30	40.70	39.20
jhead	**21.00**	**21.00**	**21.00**	**21.00**	**21.00**	**21.00**	**21.00**	**21.00**	**21.00**	**21.00**

The comparison to other coverage metrics. In this experiment, we compare our proposed metrics with two other coverage metrics for input prioritization: AFL-Sensitive [53] and LEOPARD [15].

AFL-Sensitive [53] presents 7 coverage metrics such as memory access address (memory-access-aware branch coverage) and n-basic block execution path (n-gram branch coverage). We ran our metrics and the 7 coverage metrics of AFL-Sensitive on the same testing suite and equal amount of time reported in the paper [53], and we compared them with regard to the number of discovered vulnerabilities and code coverage.

Table IX and Table X show the number of discovered vulnerabilities and the code coverage associated with the metrics of coverage accounting and AFL-Sensitive. Per AFL-Sensitive [53], a metric should be included if it achieves the top of all metrics in the associated found vulnerabilities or code coverage on a target program. Based on our experiment, we see that all metrics are necessary for coverage accounting and AFL-Sensitive. Taking all metrics into account, we observed that coverage accounting reported a few more vulnerabilities than AFL-Sensitive. Coverage accounting also slightly outperformed in code coverage, given that it achieved a higher coverage in 66.7% (8/12) and a lower coverage in 16.7% (2/12) of the target programs. The results indicate coverage account performed slightly better than AFL-Sensitive in the number of discovered vulnerabilities and code coverage with fewer metrics, and that the metrics of coverage accounting are more effective than those of AFL-Sensitive.

TABLE XI: The Number of vulnerabilities found by LEOPARD and TortoiseFuzz (10 runs).

Program	TortoiseFuzz		LEOPARD	
	Average	Max	Average	Max
exiv2	9.7	12	6.0	11
new_exiv2	5.0	5	0.0	0
exiv2_9.17	1.0	1	0.7	1
gpac	7.6	9	5.8	8
liblouis	1.2	2	0.0	0
libming	3.0	3	3.0	3
libtiff	1.2	2	0.0	0
nasm	3.0	4	1.9	3
ngiflib	4.7	5	3.7	5
flvmeta	2.0	2	2.0	2
tcpreplay	1.2	2	0.0	0
catdoc	2.1	3	2.0	2
SUM	41.7	50	25.1	35
p-value of the Mann-Whitney U test			0.0001707	

LEOPARD [15] proposed a function-level coverage accounting scheme for input prioritization. Given a target program, it first identifies potentially vulnerable functions in the program, and then it calculates a score for the identified functions. The score of a function is defined based on code complexity properties, such as loop structures and data dependency. Finally, LEOPARD prioritized inputs by the sum of the score of the potentially vulnerable functions executed by the target program with each input. On the contrary of TortoiseFuzz which prioritize inputs by basic block-level metrics, LEOPARD assesses the priority of inputs based on the level of functions and in particular, pre-identified potentially vulnerable functions.

Since LEOPARD integrates metrics such as code complexity and vulnerable functions internally, we compare the result of TortoiseFuzz as the integration of code coverage to that of the corresponding LEOPARD fuzzer. As the LEOPARD fuzzer or metric implementation is not open-sourced, we contacted the authors and received from them the identified potentially vulnerable functions with computed scores. We then wrote a fuzzer, per their suggestion of the design, deployed the computed scores to the fuzzer, and ran the fuzzer with the LEOPARD metrics. We kept the total amount of time 140 hours for each experiment, and compared the number of discovered vulnerabilities between code coverage and LEOPARD metrics, shown in Table XI.

We observe that TortoiseFuzz found more vulnerabilities than LEOPARD from 83% (10/12) applications on average and an equal number of vulnerabilities from the other 2 applications. The p-value is 0.0001707, which demonstrates statistical significance between TortoiseFuzz and LEOPARD and thus the coverage accounting metrics, which is on basic block level, perform better than the LEOPARD metrics, which is on function level, in identifying vulnerabilities from real-world applications.

TABLE XII: The number of vulnerabilities detected by QSYM with and without coverage accounting (5 runs).

Program	Vulnerabilities									
	QSYM(+AFL)		QSYM+func		QSYM+bb		QSYM+loop		QSYM+CA	
	AVG	Max	AVG	Max	AVG	Max	AVG	Max	AVG	Max
exiv2	8.2	10	11.2	12	8.4	11	10.6	13	13.0	15
new_exiv2	7.4	9	5.5	8	3.8	8	5.0	8	7.3	9
exiv2_9.17	2.0	2	1.5	2	1.5	2	1.8	3	1.8	3
gpac	6.0	8	7.0	10	8.4	10	7.6	9	9.2	11
liblouis	2.0	3	2.6	3	2.8	3	2.4	3	3.0	3
libming	3.0	3	3.0	3	3.0	3	3.0	3	3.0	3
libtiff	0.8	1	0.4	1	0.6	1	0.8	1	0.8	1
nasm	2.2	3	2.6	3	2.8	3	2.8	4	3.2	4
ngiflib	4.2	5	5.0	5	5.0	5	4.8	5	5.0	5
flvmeta	2.0	2	2.0	2	2.0	2	2.0	2	2.0	2
tcpreplay	0.0	0	0.0	0	0.4	1	0.6	1	0.6	1
catdoc	2.0	2	2.0	2	2.0	2	2.0	2	2.0	2
SUM	39.8	48	43.6	51	41.4	51	44.0	54	51.2	59
p-value of the U test			0.2477059		0.5245183		0.1387917		0.0119252	

E. RQ4. Improving the State-of-the-art with Coverage Accounting.

As an input prioritization mechanism, coverage accounting is able to cooperate with other types of fuzzing improvement such as input generation. In this experiment, we study the question that whether coverage accounting, as an extension to the state-of-art fuzzer, helps improve the effectiveness in vulnerability discovery.

Recall that QSYM was best-performed compared fuzzers in our experiment, and that it found 9 vulnerabilities that are missed by TortoiseFuzz. Therefore, we selected QSYM and compared the number of vulnerabilities found by QSYM with and without coverage metrics.

In this experiment, we compared QSYM to the other four variations: QSYM with the function metric (QSYM+func), the basic block metric (QSYM+bb), the loop metric (QSYM+loop), and with the full coverage accounting metrics (QSYM+CA). We ran all tools for 140 hours and repeated each experiment for 5 times.

Table XII shows the number of vulnerabilities discovered by each fuzzer. We find that all the metrics help to improve QSYM in vulnerability discovery. In particular, QSYM with full coverage accounting (QSYM+CA) is able to find 28.6% more vulnerabilities on average, and 22.9% more in the sum of the best per-program performance. This indicates that coverage accounting is able to cooperate with the state-of-the-art fuzzers and significantly improve the effectiveness in vulnerability discovery.

F. RQ5: Defending against Anti-fuzzing

Recent work [23, 28] show that current fuzzing schemes are vulnerable to anti-fuzzing techniques. Fuzzification [28], for example, proposes three methods to hinder greybox fuzzers and hybrid fuzzers. Fuzzification effectively reduced the number of discovered paths by 70.3% for AFL and QSYM on real-world programs.

To test the robustness of coverage accounting against Fuzzification, we implemented TortoiseFuzz-Bin, a version of TortoiseFuzz to test binary programs based on AFL Qemu mode and IDA Pro. We got 4 testing binaries from Fuzzification compiled with all three anti-fuzzing methods. In our experiment, we ran TortoiseFuzz-Bin for 72 hours aligned with the setup of Fuzzification. We selected the initial seeds based on the suggestion of the author of Fuzzification. We also used Fuzzification's method to get the numbers of the path discovered. Additionally, we measured the code coverage of TortoiseFuzz-Bin and AFL.

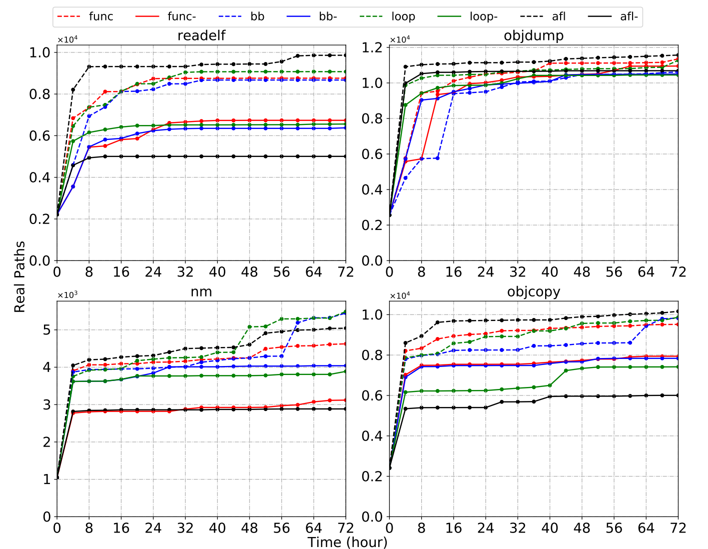

Fig. 6: Paths discovered by TortoiseFuzz from real-world programs. Each program is compiled with eight settings: func, bb, loop, afl (without protection), func-, bb-, loop-, afl-(with all protections of Fuzzification).

Table XIII and Table show the number of discovered paths and the code coverage of each testing cases after 72 hours of fuzzing process. Based on the tables, we find that AFL decreased much more than all the metrics in coverage accounting. Additionally, we did the statistics of number of discovered paths over time, shown in Figure 6. The figure indicates that the coverage accounting metrics consistently performed better than AFL since 4 hours after the experiment starts, which indicates that coverage accounting is more robust than AFL over time.

VII. Discussion

A. Coverage Accounting Metrics

TortoiseFuzz prioritizes inputs by a combination of coverage and security impact. The security impact is represented by the memory operations on three different types of granularity at function, loop, and instruction level. These are empirical heuristics inspired by Jia et al. [27]. We see our work as a first step to investigate how to comprehensively account coverage quantitatively and adopt the quantification to coverage-guided fuzzing. In the future, we plan to study the quantification in a more systematic way. A possible direction is to consider more heuristics and apply machine learning to recognize the feasible features for effective fuzzing.

TABLE XIII: The number of discovered paths of the Anti-fuzz experiment. Each program is compiled with eight settings: func, bb, loop, afl (without Fuzzification protection), func-, bb-, loop-, afl- (with full Fuzzifucation protection). The blue values denote the highest decrease rate.

Program	#Paths discovered							
	TortoiseFuzz-Bin						AFL-Qemu	
	func	func-	bb	bb-	loop	loop-	afl	afl-
nm	4626	3120 (-32.5%)	5449	4040 (-25.8%)	5490	3888 (-29.1%)	5043	2884 (**-42.8%**)
objcopy	9518	7942 (-16.5%)	9823	7837 (-20.2%)	9869	7417 (-24.8%)	10165	6005 (**-40.9%**)
objdump	11353	10951 (-3.5%)	10618	10508 (-1.0%)	11272	10448 (-7.3%)	11569	10688 (**-7.6%**)
readelf	8761	6737 (-23.1%)	8673	6378 (-26.4%)	9075	6560 (-27.7%)	9863	5006 (**-49.2%**)

TABLE XIV: The code coverage result of the Anti-fuzz experiment. Each program is compiled with eight settings: func, bb, loop, afl (without Fuzzification protection), func-, bb-, loop-, afl- (with full Fuzzification protection). The numbers in parentheses are the decrease rates cased by Fuzzification. The blue values denote the highest decrease rate.

Program	Code coverage							
	TortoiseFuzz-Bin						AFL-Qemu	
	func	func-	bb	bb-	loop	loop-	afl	afl-
nm	5.2%	4.0% (-1.2%)	5.2%	4.8% (-0.4%)	5.3%	4.5% (-0.8%)	5.3%	3.9% (**-1.4%**)
objcopy	8.2%	7.5% (-0.7%)	8.3%	7.4% (-0.9%)	8.5%	7.0% (-1.5%)	8.8%	6.2% (**-2.6%**)
objdump	7.5%	7.3% (-0.2%)	7.3%	7.3% (-0.0%)	7.5%	7.5% (-0.0%)	7.8%	7.4% (**-0.4%**)
readelf	18.9%	15.8% (-3.1%)	18.4%	15.7% (-2.7%)	18.7%	15.50% (-3.2%)	20.5%	12.6% (**-7.9%**)

B. *The Threshold for Security-sensitivity*

We currently set a unified threshold for deciding security-sensitive edges: an edge is security-sensitive if one of the three metrics' value is above 0. Ideally, the threshold should be specific to programs. Future work would be to design approaches to automatically generate the threshold for each program through static analysis or during the fuzzing execution.

C. *LAVA-M vs. Real-world Data Set*

In our experiment, we observed that the LAVA-M test suite is different from the real-world program test suite in two aspects. First, LAVA-M has more test cases involving magic words, which makes the test suite biased in testing exploit generation tools. Second, the base binaries for LAVA-M are not as complex as the real-world programs we tested. We acknowledge the value of the data set; yet a future direction is to systematically compare the inserted bugs in LAVA-M against a large number of real-world programs, understand the difference and the limitation of the current test suite, and build a test suite that is more comprehensive and more representative to real-world situations.

D. *Statistical Significance of #Vulnerabilities per Program*

In our evaluation, we did not report the statistical significance between TortoiseFuzz and other fuzzers in terms of the number of vulnerabilities found per program. On the contrary to the per-program p-value in code coverage shown in Table VI and represented in REDQUEEN [4], vulnerabilities are very sparse in one vulnerabilities. Therefore, it is hard to tell the effectiveness difference among fuzzers from a single program. For example, for the `flvmeta` program, the statistical significance among all tools are inconclusive, since all tools found 2 vulnerabilities across all runs. Therefore, we report the p-value of the total number of vulnerabilities found from all target programs. This also indicates the necessity of having a comprehensive data suite with a sufficient number of vulnerabilities.

VIII. Related Work

Plenty of techniques have been presented to improve fuzzing in different aspects since the concept of fuzzing was developed in 1990s [38]. In this section, we introduce some of the presented fuzzing techniques. For a more comprehensive study on fuzzing techniques, please refer to recent surveys such as Chen et al. [10], Li et al. [32], and Manès et al [37].

Fuzzing specific program types. Some fuzzing techniques focus on specific types of programs, based on which they propose more effective fuzzing techniques. These studies include fuzzing on protocol [5, 14], firmware [11, 17, 39], and OS kernel [13, 45, 51, 57].

Hardware-assisted fuzzing. Previous work proposes various approaches to increase the efficiency of the fuzzing process. Xu et al. [56] designs three new operating primitives to remove execution redundancy and improve the performance of AFL in running testing inputs. Hardware-based fuzzing, such as kAFL [45] and PTfuzz [61], leverages hardware features such as Intel Processor Trace [25] to guide fuzzing without the overhead caused by instrumentation. These techniques decreases the time spent on program execution and information extraction, so that fuzzers can explore more inputs within a given amount of time.

Generational fuzzing. Besides mutation fuzzing, generational fuzzers also play an important role to test programs and find security vulnerabilities. Fuzzers such as Peach [16], Sulley [3], and SPIKE [2], generate samples based on a pre-defined configuration which specifies the input format. As generational fuzzers requires a format configuration which typically is generated manually, these fuzzers are less generic and depend on human efforts.

Machine learning-assisted fuzzing. Recent works, such as Skyfire [52], Learn&fuzz [22], and NEUZZ [48], combines fuzzing with machine learning and artificial intelligence. They learn the input formats or the relationships between input and program execution, and use the learned result to guide the generation of testing inputs. Such fuzzers, though, usually cause high overhead, because of the involvement of machine learning processing.

Static analysis and its assistance on fuzzing. Dowser [24] performs static analysis at compile time to find vulnerable code like loops and pointers, so as QTEP [54]. Sparks et al. [49] extracts the control flow graphs from the target to help input generation. Steelix [33] and VUzzer [43] analyze magic values, immediate values, and strings that can affect control flow.

Dynamic analysis and its assistance on fuzzing. Dynamic analysis including symbolic execution and taint analysis help enhance fuzzing. Taint analysis is capable of showing the relationship between input and program execution. BuzzFuzz [19], TaintScope [55], and VUzzer [43] use this technique to find relevant bytes and reduce the mutation space. Symbolic execution helps exploring program states. KLEE [8], SAGE [21], MoWF [42], and Driller [50] use this technique to execute into deeper logic. To solve the problems such as path explosion and constraint complexity, SYMFUZZ [9] reduces the symbolized input bytes by taint analysis, while Angora [12] performs a search inspired by gradient descent algorithm during constraint solving. Another problem is that program analysis will cause extra overhead. REDQUEEN [4] leverages a lightweighted taint tracking and symbolic execution method for optimization.

IX. Conclusion

In this paper, we propose TortoiseFuzz, an advanced coverage-guided fuzzer with a novel technique called *coverage accounting* for input prioritization. Based on the insight that the security impact on memory corruption vulnerabilities can be represented with memory operations, and that memory operations can be abstracted at different levels, we evaluate edges based on three levels, function call, loop, and basic block. We combine the evaluation with coverage information for input prioritization. In our experiments, we tested TortoiseFuzz with 6 greybox and hybrid fuzzers on 30 real-world programs, and the results showed that TortoiseFuzz outperformed all but one hybrid fuzzers, yet spent only 2% of memory resources. Our experiments also showed that coverage accounting was able to defend against current anti-fuzzing techniques. In addition, TortoiseFuzz identified 20 zero-day vulnerabilities, 15 of which have been confirmed and released with CVE IDs.

Acknowledgement

We thank the anonymous reviewers of this work for their helpful feedback. We also thank Peng Chen, Xiaoning Du, Jinho Jung, Cornelius Aschermann and other authors of Angora, LEAPORD, Fuzzification and Redqueen for their help with experiments. This research is supported, in part, by National Natural Science Foundation of China (Grant No. U1736209), Peng Cheng Laboratory Project of Guangdong Province PCL2018KP004. Dinghao Wu's research was supported in part by the PNC Technologies Career Development Professorship. All opinions expressed in this paper are solely those of the authors.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: Principles, techniques, and tools," *Addison wesley*, 1986.

[2] D. Aitel, "An introduction to spike, the fuzzer creation kit," *presentation slides, Aug*, 2002.

[3] P. Amini and A. Portnoy, "Sulley fuzzing framework," http://www.fuzzing.org/wp-content/SulleyManual.pdf, [2019-6-1].

[4] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[5] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a stateful network protocol fuzzer," in *Proceedings of the 9th International Conference on Information Security*. Springer-Verlag, 2006.

[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*. ACM, 2017.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[8] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008.

[9] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

[10] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, 2018.

[11] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *NDSS*, 2018.

[12] 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.

[13] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[14] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association, 2015.

[15] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: Identifying vulnerable code for vulnerability assessment through program metrics," in

Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 2019.
- [16] M. Eddington, "Peach fuzzing platform," https://www.peach.tech/products/peach-fuzzer/peach-platform/, [2019-6-1].
- [17] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.
- [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] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.
- [20] gcov, "a test coverage program," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov, fetched 2020.
- [21] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Queue*, 2012.
- [22] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017.
- [23] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "AntiFuzz: Impeding fuzzing audits of binary executables," in *Proceedings of the 28th USENIX Security Symposium*, 2019.
- [24] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22Nd USENIX Conference on Security*. USENIX Association, 2013.
- [25] Intel, "Processor Tracing," https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.
- [26] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "TIFF: Using input type inference to improve fuzzing," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018.
- [27] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards efficient heap overflow discovery," in *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 2017.
- [28] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *Proceedings of the 28th USENIX Security Symposium*, 2019.
- [29] 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*. ACM, 2018.
- [30] 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.
- [31] 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.
- [32] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, 2018.
- [33] 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.
- [34] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: An automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016.
- [35] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.
- [36] 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*. USENIX Association, 2019.
- [37] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *CoRR*, 2018.
- [38] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Comm. ACM*, 1990.
- [39] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
- [40] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.
- [41] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.
- [42] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016.
- [43] 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*. The Internet Society, 2017.
- [44] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, 2014.
- [45] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.
- [46] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity

checker," in *Usenix Conference on Technical Conference*, 2012.
[47] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," IEEE, 2016.
[48] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program smoothing," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE, 2018.
[49] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007.
[50] 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*. The Internet Society, 2016.
[51] D. Vyukov, "syzkaller," https://github.com/google/syzkaller, fetched 2020.
[52] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.
[53] 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 (RAID 2019)*. USENIX Association, 2019.
[54] S. Wang, J. Nam, and L. Tan, "QTEP: Quality-aware test case prioritization," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017.
[55] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010.
[56] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.
[57] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *IEEE Symposium on Security and Privacy*, 2019.
[58] 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 *ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery*. IEEE, 2019.
[59] 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.
[60] M. Zalewski, "American fuzzy lop (AFL) fuzzer," http://lcamtuf.coredump.cx/afl/technical_details.t, 2013.
[61] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, 2018.