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

Abstract—**Coverage-based fuzzing has been actively studied and widely adopted for finding vulnerabilities in real-world software applications. With code coverage, 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, which aims to 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 patched 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 real-world vulnerabilities.**

To resolve the issue, we propose *coverage accounting*, an innovative approach that evaluates code coverage by security impacts. Based on the proposed metrics, we design a new scheme to prioritize fuzzing inputs and develop TortoiseFuzz, a greybox fuzzer for memory corruption vulnerabilities. We evaluated TortoiseFuzz on 30 real-world applications and compared it with 5 state-of-the-art greybox and hybrid fuzzers (AFL, AFLFast, FairFuzz, QSYM, and Angora). TortoiseFuzz outperformed all greybox fuzzers and most hybrid fuzzers. It also had comparative results for other hybrid fuzzers yet consumed much fewer resources. Additionally, TortoiseFuzz found 18 new real-world vulnerabilities and has got 8 new CVEs so far. Also, coverage accounting is able to defend against current anti-fuzzing techniques effectively. We will open source TortoiseFuzz to foster future research.

I. Introduction

Fuzzing has been extensively used to find real-world software vulnerabilities. Companies such as Google and Apple have deployed fuzzing techniques to discover vulnerabilities, and researchers have proposed various fuzzing techniques [5, 7, 8, 15, 20, 34, 39, 48, 50, 52, 61, 65, 66]. Specifically, coverage-guided fuzzing [5, 7, 15, 20, 39, 48, 50, 61, 66] has been actively studied in recent years. In contrast to generational fuzzing, which generates inputs based on given format specifications [3, 4, 18], coverage-guided fuzzing does not require knowledge such as input format or programs specifications. Instead, coverage-guided fuzzing mutates inputs randomly and uses coverage to select and prioritize mutated inputs.

AFL [65] leverages edge coverage (a.k.a. branch coverage or transition coverage), and libFuzzer [52] 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 [15], memory access address, and more preceding basic blocks [58].

However, previous work treats all edges equally, neglecting that the likelihoods of edge destinations being vulnerable are different. As a result, inputs executing "cold paths" are treated as important as the others and are selected for mutation and fuzzing. Although such design is reasonable for dynamic software testing, which aims to full program coverage, it delays the discovery of a vulnerability. Even worse, such nondiscriminatory design can be attacked by recent anti-fuzzing techniques [26, 33] and becomes much less effective in finding vulnerabilities.

Therefore, we need a new input prioritization method that is able to prioritize inputs that are more likely to trigger memory corruption vulnerabilities. CollAFL [20] 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. VUzzer [48] helps to mitigate the issue by de-prioritizing inputs that lead to error-handling code, but it depends on taint analysis, which is expensive and also vulnerable to current anti-fuzzing techniques.

In this paper, we propose *coverage accounting*, a new approach for input prioritization. Our insight is that, any work that add 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. Memory corruption vulnerabilities are closely related to sensitive memory operations, and sensitive memory operations can be represented at the granularity of a function, a loop, and a basic block. To find memory corruption vulnerabilities effectively, we should aim to cover as many edges associated with sensitive memory operations as possible, and we should *only* care about these edges. Based on the intuition, our approach assesses edges from function, loop, and basic block perspectives, and we mark edges with either security-sensitive or security-insensitive. We prioritize inputs with new security-sensitive coverage, and cull the prioritized inputs by the hit count of security-sensitive edges yet meanwhile guarantee the selected inputs cover all visited security-sensitive edges.

Based on the proposed approach, we develop TortoiseFuzz,

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.2020.23422
www.ndss-symposium.org

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. The implementation of TortoiseFuzz is merely about 1400 lines of code. TortoiseFuzz is simple yet powerful in finding vulnerabilities. We evaluated TortoiseFuzz on 12 popular real-world applications and compared TortoiseFuzz with 5 state-of-the-art greybox [8, 37, 65] and hybrid fuzzers [15, 64]. TortoiseFuzz outperformed all fuzzers in terms of the total number of discovered vulnerabilities, while it costed much less memory resources than hybrid fuzzers. Particularly, TortoiseFuzz found 18 zero-day vulnerabilities, 8 of which have been confirmed with CVE IDs.

We also measured the code coverage of TortoiseFuzz and compared it with the tools mentioned above. We observed that TortoiseFuzz had a similar amount of code coverage yet costed much fewer resources and found more vulnerabilities. This finding implies that the proposed coverage accounting approach is effective in exploring programs and finding memory corruption vulnerabilities.

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

- We propose *coverage accounting*, a novel approach for input prioritization with metrics that evaluate edges in terms of the relevance of memory corruption vulnerabilities. Our approach is able to distinguish edges for coverage-driven fuzzers, guarantees that the prioritized inputs cover all security-sensitive edges, and costs low overhead without expensive analysis such as taint analysis and symbolic execution.
- We design and develop TortoiseFuzz, a greybox fuzzer based on the proposed input prioritization scheme.
- We evaluated TortoiseFuzz on 12 real-world programs and compared with 5 state-of-the-art greybox and hybrid fuzzers. TortoiseFuzz outperformed the 5 fuzzers in finding more vulnerabilities, and it costed much fewer resources than hybrid fuzzers. TortoiseFuzz also successfully found 18 zero-day vulnerabilities, 8 of which have been confirmed with CVE IDs. We will open source our tool and the experiment data set.

II. Background

In this section, we present the background of input prioritization in coverage-guided fuzzing. We first introduce the high-level design of coverage-guided fuzzing, then we explain the details of input prioritization and input mutation in fuzzing. Lastly, we present an example to show the complexity of memory errors and discuss the limitation of current fuzzers.

A. *Coverage-guided Fuzzing*

Fuzzing is an automatic program testing technique for generating and testing inputs to find software bugs [43]. It is flexible and easy to apply to different programs, as it does not require the understanding of programs or manually generating testing cases.

[1]The name comes from a story of Aesop's Fables. The tortoise who is ridiculed by a hare at first crawls slowly but steadily, wins the race finally. As American fuzzy lop is a kind of rabbit, TortoiseFuzz is expected to win.

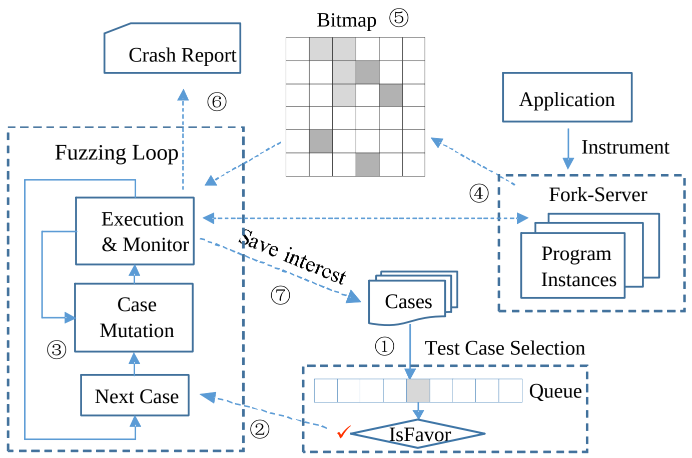


Fig. 1: The framework of AFL.

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

Figure 1 shows the architecture of AFL [65], a reputable coverage-guided fuzzer based on which a high number of 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 with a testcase, the fuzzer marks and reports the sample as a proof of concept of a vulnerability (⑥). If the testcase is interesting under the metrics, the fuzzer puts the sample into the queue and labels it as a favored sample if it satisfies the condition of being favored (⑦).

B. *Input Prioritization*

Input prioritization (a.k.a., seed selection) 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 test coverage, including block coverage, edge coverage, path coverage, etc. Comparing to block coverage, edge coverage is more delicate and more sensitive, as it takes into account the transition between blocks. Edge coverage is 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 $\textcircled{7}$ in Figure 1) and queue culling (step $\textcircled{1}$ 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 it 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 one 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. 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 hash collision. CollAFL [20] 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 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 firstly 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 prioritization 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 it will 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.

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 considers finer-grained context such as call frame. Under such circumstance, previous work proposes to resolve the issue by including more information to coverage representation. Angora [15] proposes to add calling stack, and Wang et al. [58] 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 the typical edge coverage to be more specific, 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 [48] de-prioritizes the inputs that leading to error handling codes or frequent paths. However, it requires extra heavy-weight work to identify error-handling codes, which makes fuzzing less efficient.

CollAFL [20] 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 string assignment.

C. *Anti-Fuzzing Techniques*

As more studies on fuzzing, anti-fuzzing techniques [26, 33] are proposed. They target coverage-guided fuzzers from two perspectives: 1) most coverage-guided fuzzers do not differentiate the coverage of different edges, and 2) hybrid fuzzers use heavy-weighted taint analysis or symbolic execution. Anti-fuzzing techniques fools fuzzers by inserting fake paths, adding a delay in cold paths, and obfuscating codes to slow down dynamic analyses.

Current anti-fuzzing techniques make coverage-guided fuzzers much less effective in vulnerability discovery, causing the fuzzers 85%+ performance decrease in exploring paths. Unfortunately, all of the presented edge-coverage-based fuzzers [8, 15, 58, 65] suffer from the current anti-fuzzing techniques. Moreover, any approach that adds more information to edge representation will not work well under the presence of anti-fuzzing. Essentially, this is because that edge coverage is treated equally despite the fact that edges have different likelihoods to lead to vulnerabilities.

D. *Input Mutation*

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 [15, 31, 37, 63] and optimal mutation scheduling approaches [41]. These input mutation approaches are all complementary to our proposed input prioritization scheme.

III. Coverage Accounting

Prior coverage-guided fuzzers [5, 7, 8, 15, 39, 50, 52, 61, 65, 66] 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 [26, 33] 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 light-weighted. 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 [32], 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 [20] counts the total number of memory access operations throughout the execution path, which implies the *history* memory access operations. Wang et al. [58] applies the address rather than the count of memory access. Type-aware fuzzers such as Angora [15], TIFF [31], and ProFuzzer [63] 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, and we should prioritize the inputs that will visit this function. As functions in the call stack of a crash is not equivalent to vulnerable functions, we are trying to construct an input that causes a crash instead of just triggering a vulnerability. We choose the functions on the stack for input prioritization, even though we can figure out the vulnerable function for each vulnerability.

To find the vulnerability-involved functions, we go over the disclosed vulnerabilities in the latest 4 years. We crawl the reference webpages on CVE descriptions, extract the call stacks from the reference webpages and synthesize the involved functions. Part of them are shown in Table I. 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 `strcpy` and `memcpy` are more likely to be involved in memory corruptions.

TABLE I: Top 20 vulnerability involved functions

Function	Num	Function	Num
memcpy	195	free	36
memset	70	read	35
ReadImage	51	memmove	17
pnmscanner_gettoken	50	JPXStream::fillReadBuf	15
pnm_load_ascii	50	ReadNextFunctionHandle	13
pnm_load_rawpbm	50	Mat_VarReadNextInfo5	13
pnm_load_raw	50	ReadNextCell	13
input_pnm_reader	50	ReadNextStructField	13
GET_COLOR	50	Mat_VarPrint	13
strlen	37	mp4ff_read_stts	11

Given the vulnerability-involved functions, we assess an edge by the number 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:

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

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

One possible concern is that this metric cannot find vulnerabilities associated with functions that were not involved in any vulnerabilities before or custom functions. This concern is valid for the metric of function calls. However, it can be resolved by the other two metrics, under the intuition that if a function was not involved in a vulnerability before, one should investigate the memory access in a finer granularity, which are captured by the loop and basic block metrics.

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 back edge to indicate whether an input will iterate a loop. 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 [2]. Let function $\mathrm{IsBackEdge}(e)$ 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 $\mathrm{Loop}(e)$ as follows:

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

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 $\mathrm{IsContainMem}(i)$ 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 $\mathrm{BB}(e)$ as follows:

$$\mathrm{BB}(e) = card\Big(\big\{i | i \in dst_e \wedge \mathrm{IsContainMem}(i)\big\}\Big) \tag{3}$$

Discussion: Coverage accounting for basic blocks. Coverage accounting is also applicable for basic blocks, as the three above metrics can be equivalently converted from an edge to its destination basic block. Coverage accounting on basic blocks is weighted though, where the weight is equal to the number of incoming edges. The weighted result of block coverage accounting is mathematically equivalent to that of edge coverage. We refer the readers to Section VII for discussions on other coverage metrics.

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. Recall that 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 matters 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 second challenge is how to prioritize an input that are more likely to evolve to trigger a vulnerability, based on the proposed metrics for coverage accounting. Our intuition is that, the more an inputs hits a security-sensitive edge, the more likely the input will evolve to trigger a vulnerability. Based on this intuition, we use hit count as the metrics for prioritizing inputs associated with a same edge.

The last 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 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).

Based on the above considerations, we decide to design TortoiseFuzz based upon AFL, and we 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 light-weighted 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 the 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 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 triggers new edge coverage (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*).

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

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 previous saved one, we mark it as "favor" if the cost is less. The *AccountingFb* and *accAccounting* are determined by coverage accounting.

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

2) *Queue Culling*

The top-rated candidates recorded by *TopAccounting* are a superset of samples that can cover all the memory-related 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. First we create and initialize 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 greedy 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 less 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 memory-sentitve 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] && !Temp_map[i].favor == 1 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].favor == 1 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 cold paths, 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 greatly affected by anti-fuzzing because input prioritization helps to avoid the cold paths, and fake branches created by Fuzzification [33] with `pop` and `ret` are unlikely to be prioritized.

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 [65]. Besides the AFL original implementation, TortoiseFuzz consists of about 1400 lines of code including instrumentation ($\sim$700 lines in C++) and fuzzing loop ($\sim$700 lines in C). 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 backedges. For the basic block level coverage accounting, we mark the memory access characteristics of the instructions with the *mayReadFromMemory()* and *mayWriteToMemory()* functions from LLVM [35].

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**: Is TortoiseFuzz better than previous fuzzers in finding real-world vulnerabilities?
- **RQ3**: Is coverage accounting better than previous coverage metrics?
- **RQ4**: Is coverage accounting robust against anti-fuzzing?

A. *Experiment Setup*

Experiment dataset. We synthesized 30 applications from the recent papers published from 2016 to 2018. For a fair comparison, we re-ran all fuzzers under a same environment, and we tested the fuzzers on same targets with the latest version. None of the fuzzers found any vulnerabilities from 18 out of the 30 applications; for ease of demonstration, we only show the results for the remaining 12 applications in the paper. These application include image parsing and processing libraries (libtiff, libexiv2, libexiv2-new, libexiv2-9.17, giflib), text parsing tools (catdoc, tcpreplay), assembly tool (nasm), multimedia file processing libraries (flvmeta, libming, libgpac) and language translation tool (liblouis).

One may argue that the experiment dataset is lack of other test suites such as LAVA-M. We argue 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.

Experiment environment. In our evaluation, we ran all experiments on four servers with 32 Intel(R) Xeon(R) CPU E5-2630 V3@2.40GHZ cores and 64GB RAM, which were installed with 64-bits Ubuntu 16.04.3 TLS. For each experiment of the target program, we configured each fuzzer with the same seed and dictionary set, under the same command line.

We set the test time to 140 hours, referred to prior work [20]. We also repeated all experiments for 5 times and reported the average value as the final experiment results as advised from Klee et al [34].

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 be run on our experiment dataset. We filtered tools that are not open-source or do not scale to the our test real-world programs and thus cannot be compared. The remained compared fuzzers are 3 greybox fuzzers (AFL, AFLFast and FairFuzz) and 2 hybrid fuzzers (QSYM and Angora). For detailed explanations for each fuzzer, we refer readers to the footnote in Table II.

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	
TIFF	2018	greybox	Y[3]	S	
FairFuzz	2018	greybox	Y	S	✓
T-fuzz	2018	hybrid	Y[4]	B	
QSYM	2018	hybrid	Y	S	✓
Angora	2018	hybrid	Y	S	✓
MOpt	2019	greybox	Y	S	✓[5]
Redqueen	2019	greybox	Y[3]	S	
afl-sensitive	2019	greybox	Y[3]	S	
EnFuzz	2019	hybrid	Y[3]	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] These tools are not fully open-source as they claim in papers.
[4] Some components of these tools cannot work for all binaries.
[5] We are contacting with the authors and working on it.

B. RQ1: Finding Zero-day Vulnerabilities

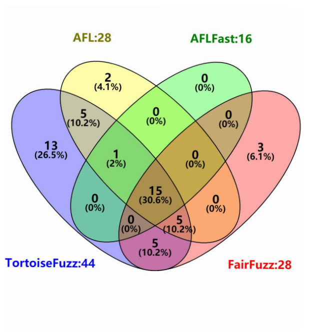
Table III shows the real-world vulnerabilities found by TortoiseFuzz from the 30 collected programs. We present each discovered vulnerability with the target program name, the identification ID, the vulnerability type and the CVSS core (Common Vulnerability Scoring System, v3.0 [1]). Also we mark the new vulnerabilities we found.

TortoiseFuzz found 44 vulnerabilities in 9 different types of common memory errors from the 12/30 vulnerabilities, including stack-buffer-overflow, heap-buffer-overflow and use-after-free. Among the found vulnerabilities, 18 of them are zero-day vulnerabilities, which is more than 40%. We validated the vulnerabilities manually and submitted vulnerability reports to the CVE officials and software developers. By the time of paper submission, we have received confirmations for 8 vulnerabilities with CVE numbers[2]. All the vulnerabilities, including previous disclosed and new ones, are assigned with medium even critical severity scores, which implies that TortoiseFuzz are capable of finding severe memory errors from real-world software programs.

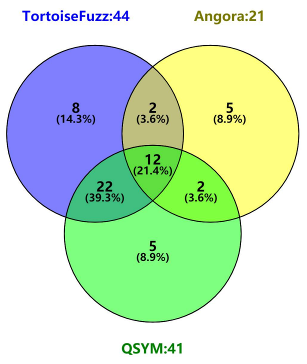
C. RQ2: Comparing TortoiseFuzz with Previous Fuzzers

We compared TortoiseFuzz with both greybox fuzzers and hybrid fuzzers in finding real-world vulnerabilities. Specifically, we evaluate each fuzzer by the number of unique crashes, the number of real-world vulnerabilities, code coverage, and performance.

Finding real-world vulnerabilities. In this experiment, we ran all fuzzers for 140 hours referred by previous work [20]. For produced crashes, we used AddressSanitizer [51] to filter the repeated crashes,and we manually identify the vulnerabilities from the crashes.



(a) Compared with greybox fuzzers.



(b) Compared with hybrid fuzzers.

Fig. 2: The intersection of the findings by different fuzzers

Table IV shows the vulnerabilities found by different fuzzers. TortoiseFuzz found 44 vulnerabilities in total, which is significantly better than the other grey-box fuzzers (AFL and FairFuzz found 28 and AFLFast found 16 vulnerabilities). Also, TortoiseFuzz found the most vulnerabilities for 67%

[2]CVE-2018-17229 and CVE-2018-17230 are contained in both exiv2 and exiv2_new.

TABLE III: Real-world Vulnerabilities found by TortoiseFuzz

Program	ID	Vulnerability Type	CVSS	New
exiv2	CVE-2018-16336	heap-buffer-overflow	6.5 MEDIUM	✓
	CVE-2018-17229	heap-buffer-overflow	6.5 MEDIUM	✓
	CVE-2018-17230	heap-buffer-overflow	6.5 MEDIUM	✓
	issue_400	heap-buffer-overflow	-	✓
	CVE-2017-11336	heap-buffer-overflow	6.5 MEDIUM	-
	CVE-2017-11337	invalid free	6.5 MEDIUM	-
	CVE-2017-14858	heap-buffer-overflow	5.5 MEDIUM	-
	CVE-2017-14861	stack-buffer-overflow	5.5 MEDIUM	-
	CVE-2017-14865	heap-buffer-overflow	5.5 MEDIUM	-
	CVE-2017-14866	heap-buffer-overflow	5.5 MEDIUM	-
	CVE-2017-17669	heap-buffer-overflow	5.5 MEDIUM	-
	CVE-2018-10999	heap-buffer-overflow	6.5 MEDIUM	-
exiv2_new	CVE-2018-17229	heap-buffer-overflow	6.5 MEDIUM	✓
	CVE-2018-17230	heap-buffer-overflow	6.5 MEDIUM	✓
	CVE-2017-14865	heap-buffer-overflow	5.5 MEDIUM	-
	CVE-2017-14866	heap-buffer-overflow	5.5 MEDIUM	-
	CVE-2017-14858	heap-buffer-overflow	5.5 MEDIUM	-
exiv2_9.17	CVE-2018-17282	null pointer dereference	6.5 MEDIUM	✓
nasm	CVE-2018-8883	buffer over-read	7.8 HIGH	-
	CVE-2018-16517	null pointer dereference	5.5 MEDIUM	-
	CVE-2018-19213	memory leaks	5.5 MEDIUM	-
gpac	issue_1183	memory leaks	-	✓
	issue_1179	Segment Fault	-	✓
	issue_1180	heap-buffer-overflow	-	✓
	issue_1077	use-after-free	-	-
	issue_1090	double-free	-	-
	issue_1188	heap-buffer-overflow	-	-
libtiff	CVE-2018-15209	heap-buffer-overflow	8.8 HIGH	✓
	CVE-2018-16335	heap-buffer-overflow	8.8 HIGH	✓
liblouis	issue_315	memory leaks	-	-
ngiflib	issue_10	stack-buffer-overflow	-	✓
	issue_11	heap-buffer-overflow	-	✓
	issue_12	heap-buffer-overflow	-	✓
	CVE-2018-11575	stack-buffer-overflow	9.8 CRITICAL	-
	CVE-2018-11576	heap-buffer-over-read	9.8 CRITICAL	-
libming	CVE-2018-13066	memory leaks	7.5 HIGH	-
	(2 similar crashes)	memory leaks	-	-
catdoc	CVE-2017-11110	heap-buffer-underflow	7.8 HIGH	-
	crash	Segment Fault	-	-
tcpreplay	CVE-2018-20552	heap-buffer-overflow	7.8 HIGH	✓
	CVE-2018-20553	heap-buffer-overflow	7.8 HIGH	✓
flvmeta	issue_13	null pointer dereference	-	✓
	issue_12	heap-buffer-overflow	-	-

(8/12) of the target programs, which we highlighted in blue. To investigate the found vulnerabilities in the specification, we built a Venn diagram to show the overlap of the findings in Figure 2. As we can see from Figure 2a that, TortoiseFuzz nearly covered all bugs found by AFL, AFLFast, and FairFuzz.

In terms of hybrid fuzzers, we followed the steps of Angora's manual to compile all programs in the real-world test set, yet one program failed to run. For the rest of the testing programs, TortoiseFuzz found more real vulnerabilities than Angora. We had contacted to the authors and analyzed that it might be caused by the limitation of DFSan [40]. QSYM found 41 vulnerabilities in total, which is close to TortoiseFuzz's findings. Among these vulnerabilities, TortoiseFuzz found 10 exclusively and QSYM found 8 exclusively. Therefore, although QSYM has less found vulnerabilities, we consider it as comparable to TortoiseFuzz in terms of real-world vulnerability discovery (see Section VI-C for further

TABLE IV: Vulnerabilities found by different fuzzers

Program	version	uniq crashes	vulnerabilities		Grey-box Fuzzers				Hybrid Fuzzers		vulnerabilities	
			unknown	known	TortoiseFuzz	AFL	AFLFast	FairFuzz	Angora	QSYM	new CVE	old CVE
exiv2	0.26	1966	4	9	**12**	10	2	8	10	9	3	12
exiv2_new	0.26	52	2	3	5	0	0	0	0	**9**	2	5
exiv2_9.17	0.26	380	1	0	1	1	1	1	0	**2**	1	1
gpac	0.7.1	8488	3	3	**6**	4	3	5	3	5	0	0
liblouis	3.7.0	828	0	1	1	0	0	0	0	**2**	0	0
libming	0_4_8	1485	0	3	**3**	3	3	3	3	3	0	1
libtiff	4.0.9	15	2	0	**2**	0	0	0	0	1	2	0
nasm	2.14rc4	7497	0	1	**3**	2	2	2	1	2	0	3
ngiflib	0.4	1363	3	2	**5**	5	2	5	2	4	0	2
flvmeta	1.2.1	579	1	1	**2**	2	2	2	2	2	0	0
tcpreplay	4.3	390	2	0	**2**	0	0	0	-	0	2	0
catdoc	0_95	84	0	2	**2**	1	1	2	0	2	0	1
SUM	-	23127	18	25	44	28	16	28	21	41	10	25

- 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".

discussion).

Code coverage. In addition to the number of bugs, we also measured the code coverage for different fuzzers. Although high code coverage is not the goal of our paper, measuring code coverage is still meaningful, as it is an import metrics for evaluating program testing techniques. More importantly, we want to evaluate if the impact-based evaluation approach impacts the overall code coverage. We used gcov [22] to obtain the code coverage of real-world applications.

TABLE V: Code coverage of different fuzzers

Program	Grey-box Fuzzers				Hybrid Fuzzers	
	TortoiseFuzz	AFL	AFLFast	FairFuzz	Angora	QSYM
exiv2	19.40%	21.30%	6.80 %	20.30%	**24.10%**	22.00%
new_exiv2	21.10%	20.10%	6.80 %	19.30%	8.70 %	**22.70%**
exiv2_9.17	19.30%	19.20%	18.80%	21.20%	8.00 %	**23.90%**
gpac	3.90 %	4.80 %	4.00 %	**6.30 %**	5.70 %	5.70 %
liblouis	29.90%	29.00%	29.70%	24.90%	24.90 %	**30.20%**
libming	21.00%	20.80%	**21.10%**	21.00%	20.20%	**21.10%**
libtiff	**43.30%**	40.10%	36.60%	38.30%	39.60%	39.70%
nasm	17.70%	30.70%	31.90%	**32.30%**	8.10 %	32.00%
ngiflib	**79.00%**	**79.00%**	**79.00%**	74.80%	**79.00%**	**79.00%**
flvmeta	**12.20%**	12.10%	12.10%	**12.20%**	12.10%	12.10%
tcpreplay	**12.70%**	12.50%	10.90%	11.00%	-	**12.70%**
catdoc	50.00%	49.80%	49.80%	50.50%	47.10%	**62.90%**
Average	27.46%	28.28%	25.63%	28.12%	25.23%	30.33%

Table V reports the line coverage of all fuzzers on the real-world programs. For 75% (9/12) of the real world programs, TortoiseFuzz could achieves higher coverage than AFLFast and Angora. The average coverage of TortoiseFuzz was close to AFL and FairFuzz, and a little less than QSYM. However, TortoiseFuzz got higher coverage than QSYM on *libtiff* and *flvmeta*, and achieved higher coverage than AFL and FairFuzz on *new_exiv2*, *libming*, *libtiff*, *ngiflib*, *flvmeta* and *tcpreplay*.

Performance. We recorded the resource usage logs of QSYM, Angora and the three strategies of TortoiseFuzz every five seconds. As shown in Figure 3, for all the real-world programs, QSYM and Angora took much more virtual memory resource than TortoiseFuzz. The reason is that the hybrid fuzzers such as QSYM and Angora need more resources to execute heavy-weighted analyses such as taint analysis and constraints solving.

Case study. During our experiment, we found that some fuzzers presented better on some targets but perform worse on the others. The reason behind is closely related to the logic of the target applications such as the execution path length and memory operations. We studied the differences case by case and try to make it clear.

Missed vulnerabilities: There are 8 vulnerabilities we missed but found by QSYM. We analyzed the cases and found that parts of the vulnerabilities are protected by conditional branches related with the input file, and the rest of them are caused by the different input prioritization strategies between TortoiseFuzz and AFL. We acknowledge the effectiveness of QSYM for binaries with input-related branches, and we plan to improve TortoiseFuzz in these cases as future work.

Unique vulnerabilities: We also 10 vulnerabilities exclusively found by TortoiseFuzz. We analyzed the case generation process for each use case and use CVE-2018-15209 in libtiff as an example. Figure 4 shows the TortoiseFuzz's search process for the CVE-2018-15209 of the original AFL and TortoiseFuzz. The *Seed ID* indicates the index of the test case generated by each fuzzer, *memr*, *memw*, and *cov* indicate that the test case is added to the TortoiseFuzz's or AFL's testcase queue based on what conditions are met. Among them, *memr* and *memw* indicate the testcase triggers the basic block containing new memory read or write operations, while *cov* indicates that the testcase triggering a new edge of the target program. *Generation* indicates the generation in which the testcase is located. For example, the initial seed is the zeroth generation, and the directly mutated from the initial one is the first generation.

The mem strategy of TortoiseFuzz generated a testcase c that triggers CVE-2018-15209 following the path $0 \rightarrow 147 \rightarrow 720 \rightarrow 1659 \rightarrow c$. In order to analyze the reason why AFL could not find this vulnerability, we tracked the testcase generation process of AFL correspondingly. The test case numbered 132 was generated by the AFL, which was the same as the testcase numbered 147 generated by the mem strategy of TortoiseFuzz. It could be seen that when the AFL used the testcase numbered 837 as the seed to generate new cases, AFL could not find any new edge was triggered or the hit count of any existed edge was increased. So the test case which was equivalent to testcase 1659 of TortoiseFuzz was not selected

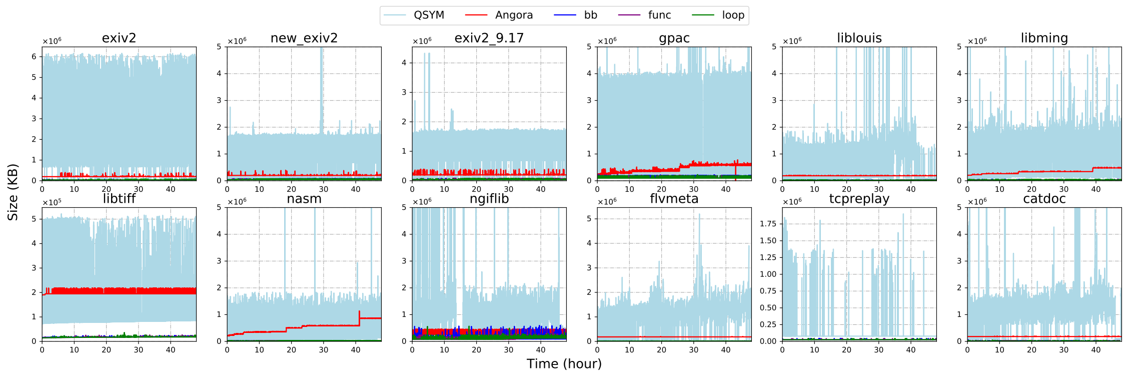


Fig. 3: Memory resource usage of QSYM, Angora and TortoiseFuzz

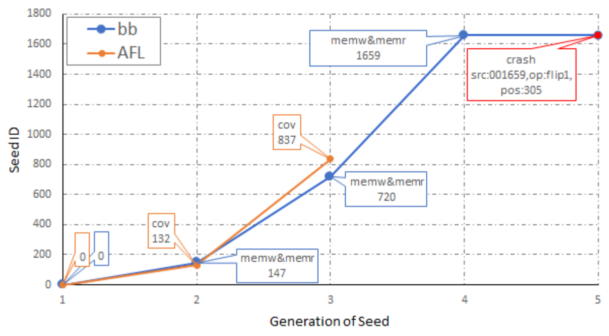


Fig. 4: Test case generation process of CVE-2018-15209

by AFL. While the mem strategy of TortoiseFuzz found that a new memory read and write behavior were triggered when it mutated the testcase 720. TortoiseFuzz named it as 1659 and finally found CVE-2018-15209 through mutating test case 1659. AFL did not find any crash during the test period with a total duration of 140 hours.

Through the analysis of the testcase generation process for CVE-2018-15209, we observe that the strategy provided by TortoiseFuzz is more directional to memory errors than AFL and other fuzzers guided by basic block or transition coverage. Although the evolution step is small, the change could be enlarged during fuzzing and TortoiseFuzz could continuously guide the detection direction towards the code segment with high-risk memory operations.

Listing 1 shows the code snippet of CVE-2017-11336, an exiv2 bug found by loop strategy of TortoiseFuzz. This is a heap-buffer-overflow bug in Line 12 and triggered by an unexpected large `offset`, which is generated during the loop process in Line 1 and Line3. The `k` value will increase continuously until the value of `k*size` is larger than `count` or the end of the heap. As the loop strategy tends to explore paths with more loops, it is easy for our tool to find this bug.

Also, our function call metrics found some unique vulnerabilities, one of them is issue_1077, a use-after-free bug in gpac. Because the metrics will generate seeds which triggered specific functions(malloc included) as much as possible, our tool is able to find this kind of bugs much faster.

```
1  for ( size_t k = 0 ; k < count ; k++ ) {
2     size_t restore = io.tell();
3     uint32_t offset = byteSwap4(buf,k*size,bSwap);
4     printIFDStructure(io,out,option,offset,bSwap,c,
          depth);
5     io.seek(restore,BasicIo::beg);
6  }
7
8  uint32_t Image::byteSwap4(DataBuf& buf,size_t offset,
       bool bSwap)
9  {
10    uint32_t v;
11    char* p = (char*) &v;
12    p[0] = buf.pData_[offset]; // buffer-overflow
13    p[1] = buf.pData_[offset+1];
14    p[2] = buf.pData_[offset+2];
15    p[3] = buf.pData_[offset+3];
16    return Image::byteSwap(v,bSwap);
17 }
```

Listing 1: The unique vulnerability found by the loop strategy of TortoiseFuzz

D. RQ3: Comparing Coverage Accounting with Previous Coverage Metrics

In this experiment, we study the 3 different metrics in coverage accounting, and we compare coverage accounting with other coverage metrics.

TABLE VI: Comparison of the three strategies of TortoiseFuzz

Program	coverage			vulnerabilities		
	func	bb	loop	func	bb	loop
exiv2	14.60%	6.40%	**19.00%**	5	4	**12**
new_exiv2	18.20%	19.70%	**20.00%**	**5**	0	0
exiv2_9.17	15.10%	**17.10%**	**17.10%**	**1**	**1**	0
gpac	**3.80%**	2.60%	2.70%	**6**	2	2
liblouis	**29.40%**	26.80%	25.10%	**1**	0	0
libming	20.50%	20.80%	**21.00%**	**3**	**3**	**3**
libtiff	**40.50%**	37.00%	37.10%	0	**2**	0
nasm	17.60%	17.60%	17.60%	1	**3**	1
ngiflib	79.00%	79.00%	79.00%	3	4	**5**
flvmeta	12.10%	12.10%	12.10%	**2**	**2**	**2**
tcpreplay	12.60%	12.50%	**12.60%**	0	0	**2**
catdoc	49.80%	49.80%	49.80%	**2**	1	1

Coverage accounting metrics. Table VI shows the result of code coverage and bug discovery of each strategy based on prioritizing different coverage accounting metrics. In the experiment, *func* represents the function call metrics, *loop*

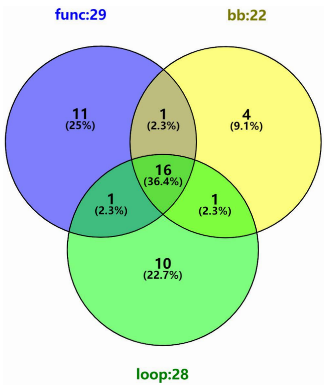


Fig. 5: The intersection of the bugs found by the different strategies of TortoiseFuzz on real-world programs

TABLE VII: Vulnerabilities detected by QSYM+TortoiseFuzz and the original QSYM

Program	vulnerabilities			
	QSYM+func	QSYM+bb	QSYM+loop	QSYM(+AFL)
exiv2	**12**	11	11	9
new_exiv2	**9**	7	8	**9**
exiv2_9.17	**2**	**2**	**2**	**2**
gpac	**5**	3	**5**	5
liblouis	**3**	**3**	2	2
libming	**3**	**3**	**3**	**3**
libtiff	0	0	0	**1**
nasm	3	3	**4**	2
ngiflib	3	**5**	4	4
flvmeta	**2**	**2**	**2**	**2**
tcpreplay	0	0	0	0
catdoc	**2**	**2**	**2**	**2**
SUM	**44**	41	43	41
TortoiseFuzz			**50**	41

represents the loop metrics, and *bb* represents the basic block metrics. For the 12 real world programs, the *loop*, *func* and *mem* metrics found 29, 22, and 28 vulnerabilities, respectively. Most of the basic blocks in each program contain memory access instructions because of compiling optimization and the effect of LLVM, and the distinction between different basic blocks is not significant. This makes the *mem* strategy nearly the same as the traditional coverage strategies used by AFL. But the *mem* strategy gets the best results in libtiff and nasm, and discovers more vulnerabilities missed by other tools. This also indicates the complexity of memory errors in different kinds of applications, and thus all metrics are valuable for finding vulnerabilities.

As for the coverage performance, *func*, *mem* and *loop* can achieve almost the same result. For *libtiff* and *gpac*, the *func* strategy achieved higher coverage than other strategies. For *exiv2*, *new_exiv2*, *exiv2_9.17*, *libming* and *tcpreplay*, the *loop* strategy achieved the highest coverage.

Figure 2 uses venn diagrams to report the logical relationships of real-world vulnerabilities that found by different strategies of TortoiseFuzz. We can see that, *func*, *bb* and *loop* strategies found a large number of different unique bugs. In other words, the result of Figure 5 reports the fact that different strategies of TortoiseFuzz can help us found different bugs effectively.

Coverage accounting + QSYM. We set an experiment to combine *loop*, *func* and *bb* strategies with QSYM and show the results in Table VII. Each of the strategies of TortoiseFuzz helps QSYM detected more vulnerabilities than the original one. The three metrics of TortoiseFuzz found 50 real vulnerabilities totally, 17 of them cannot be found by the original QSYM which executed based on AFL.

Coverage accounting v.s. other metrics. WANG et. al [58] presents multiple coverage metrics such as memory access address (memory-access-aware branch coverage) and n-basic block execution path (n-gram branch coverage), and most of these metrics are widely used by current fuzzers. We compared the ability of bug detection of them based on the test suit of afl-sensitive [58] for 72 hours (set by afl-sensitive). As shown in Table VIII, although TortoiseFuzz has only three metrics, which is far less than the number of afl-sensitive (seven), TortoiseFuzz found more (20) real vulnerabilities than afl-sensitive (18).

TABLE VIII: Compare the three metrics of TortoiseFuzz with other metrics

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

E. Defending against Anti-fuzzing (RQ4)

Recent work [33] show that current fuzzing schemes are vulnerable to anti-fuzzing techniques. Fuzzification [33], for example, proposed three methods to hinder the grey-box fuzzers and hybrid fuzzers. For AFL and QSYM, Fuzzification effectively reduces the number of discovered paths by 70.3% from real-world binaries. To test the robustness of coverage accounting against Fuzzification, we compiled the test suit provided by Fuzzification with all three anti-fuzzing methods (the build scripts are also provided by the authors of Fuzzification) and tested them for 24 hours. As shown in Figure 6, while previous work drops performance by 70%, TortoiseFuzz only decreased about 10%. This result indicates that coverage accounting is able to defend against anti-fuzzing effectively.

VII. Discussion

A. Improving Impact 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. [32]. We see our work as a first step to investigate how to comprehensively quantify security impact and adopt the quantification to coverage-guided

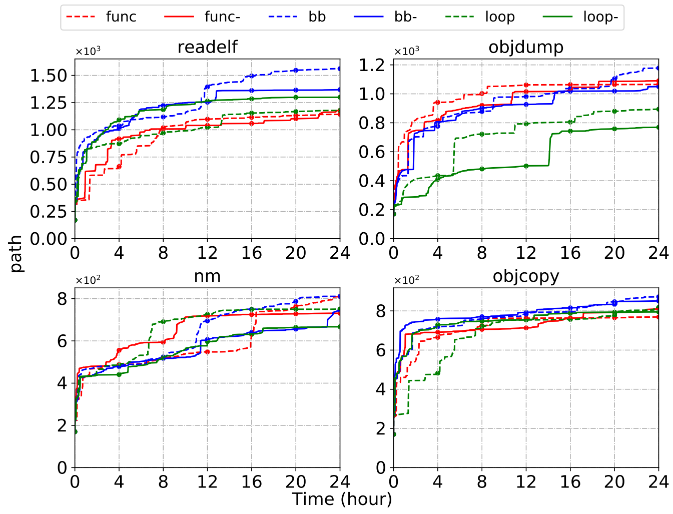


Fig. 6: Paths discovered by TortoiseFuzz from real-world programs. Each program is compiled with six settings: func, bb, loop (without protection), func-, mem-, loop-(with all protections of Fuzzification), We fuzzed them for 20 hours.

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.

B. *Identifying Sensitive Functions*

The current sensitive functions identified in TortoiseFuzz are memory-related functions in `glibc`. This design is limited for two points. First, if a binary is compiled statically and stripped, then identifying these `libc` functions are challenging and error-prone if source code is not available. Second, if a program implements its own memory operation functions, or uses a libc library that TortoiseFuzz cannot identify, then TortoiseFuzz fails and will be less effective in finding memory corruptions. To resolve these issues, we will look into more advanced techniques to identify sensitive functions. For example, we can identify sensitive functions at run time with the combination of the statistics on loop-level and instruction-level impact evaluation. Program analysis can also help identify the sensitive functions.

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. *Coverage Accounting for Other Coverage Metrics*

In this paper, we present coverage accounting for edges, and we describe the way to convert coverage accounting from edges to basic blocks. However, how to convert to other coverage metrics remains open. In the future, we will investigate the solutions to convert to other coverage metrics such as edges with additional information, and we will study how to incorporate such coverage accounting to the corresponding fuzzers.

E. *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.

VIII. Related Work

We have introduced evolution fuzzing as background and our changes in design and implement sections. There are more related works of general fuzzing and program analysis which will be introduced in this section.

A. *Improve fuzzing framework*

Since the concept of fuzzing was put forward in 1990s [43], plenty of related works are presented to improve fuzzing from different views [13, 38, 42]. According to the framework in Figure 1, some of them could be combined together to make existing fuzzers, including our TortoiseFuzz, more effective and efficient.

The quality of seeds would affect the result of fuzzing severely. The intuition was proved in Rebert's paper [49] and has become a consensus to minimize the size of the corpus and seeds.

As we mentioned in Section IV-B, we have to adapt the fuzzing framework to kinds of targets including protocol [6, 17], firmware [14, 19, 45], OS kernel [16, 50, 56, 62], etc. Also we need to deal with both source codes and binaries. Compile-time instrument like LLVM-based methods [7, 15, 20, 52] is fit for source code, while dynamic instrumentation like QEMU-based methods [25, 55] is for binaries. Extending with these instrumentation, TortoiseFuzz could handle more kinds of targets and test if there are memory vulnerabilities.

The goal of improvement for fuzzing loop is exploring program states as completely as possible, as fast as possible. Xu et al. designed three new operating primitives to increase the execution speed [61]. Hardware-based fuzzing [50, 66] is also a promising way to scale to real-world applications.

Matching file format and program checks including checksum and magic numbers is a big problem of exploring the target. On way is to use generation fuzzing. They generate samples directly based on grammars or modules with the description of special characters, such as Peach [18], Sulley [4] and SPIKE [3]. Another way is to combine with program analysis, which could find theses characters from the programs. More discussions will be put in the next sub-section.

As we claimed in the paper, we should pay more attention to evolve towards vulnerable states besides bypassing magic numbers. Dowser [27] focuses on complex pointer arithmetic instructions, while SlowFuzz [46] and Perffuzz [36] are towards resource-consuming algorithms. AFLFast [8] assigns more energy to the seeds with low frequency based on the Markov chain model of transition probability. And the following work AFLGo [7] allocates more energy on targeted vulnerable codes. Different from these works, we find memory errors are closely related to memory related behaviors and pay more attention to these operations at different levels.

Another important part of fuzzing is detection and report. Besides program crashes, both redzone-based memory error detectors and pointer-based detectors are proposed. We could insert sanitizer during compile with source code as AddressSanitizer [51], replace the memory allocator during runtime as MEDS [28], or set checks at instrument tools as Dr.Memory [9]. To deal with the crashes, we cry for an automatic tool to analyze them for exploit and repair, such as Microsoft's !exploitable plugin [44], CRAX [30] and HCSIFTER [29].

B. *Program Analysis*

Program analysis could help us to get more information from the targets and their inputs, which could help to explore program states deep and fast. Our method doesn't rely on heavy program analysis or reverse engineering. However, we could separate the analysis from fuzzing and get more knowledge before testing, which is very helpful to TortoiseFuzz.

Static analysis could help us get more knowledge of target program and testcases. Dowser [27] performs a static analysis during compile to find vulnerable codes like loops and pointers, so as QTEP [59]. Sparks's paper [54] extracts the control flow graphs from the target to help testcase generation. While Steelix [39] and VUzzer [48] focus on characters which could affect control flow such as magic values, immediate values and other characteristic strings before testing.

Dynamic analysis including symbolic execution and taint analysis could enhance fuzzing. As taint analysis could show the relationship between input and program execution, BuzzFuzz [21], TaintScope [60], VUzzer [48], etc. use this technique to find relevant bytes and reduce the mutation space. Symbolic execution could help to explore the program state space. KLEE [10], SAGE [23], MoWF [47], Driller [55], etc. use this technique to execute into deeper logic. To solve the problems such as path explosion and constraint complexity, SYMFUZZ [12] reduce the symbolized input butes by taint analysis, while Angora [15] performs a search inspired by gradient descent algorithm when solving. Another problem is that program analysis will cause extra overhead. REDQUEEN [5] leverages a lightweight taint tracking and symbolic execution method for optimization.

Recently learning and artificial intelligence are popular and combined with fuzzing, e.g., Skyfire [57], Learn&fuzz [24], NEUZZ [53]. They could learn input formats and relationships between input and program execution, guiding testcase generation to discover more paths.

Symbolic execution tools such as Mayhem [11] and KLEE [10] have proposed heuristics to prioritize paths such as symbolic memory access and symbolic instruction pointers. Those heuristics have been shown to be effective for symbolic execution; however, these heuristics are not applicable for fuzzing due to the absence of symbol information. Therefore, it is necessary and imperative to design a comprehensive scheme to evaluate the fuzzing testcases in terms of the security impact for memory corruption vulnerabilities.

IX. Conclusion

In this paper, we propose TortoiseFuzz, an advanced coverage-guided fuzzer with *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 to different levels, we evaluate edges based on three levels, function calls, loops, and basic blocks, and we combine the evaluation with coverage information for input prioritization. In our experiment, we tested TortoiseFuzz with 5 greybox and hybrid fuzzers on 30 real-world programs, and the results showed that TortoiseFuzz outperformed all but one hybrid fuzzers, yet spent only 1% of memory resources. Our experiment also shows that coverage accounting is able to defend against current anti-fuzzing techniques. In addition, TortoiseFuzz identified 18 zero-day vulnerabilities 8 of which have been confirmed and released with CVE IDs.

References

[1] "Common vulnerability scoring system," https://www.first.org/cvss/, [n. d.].

[2] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, 1986.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011.

[10] 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.

[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[12] 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.

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

[14] 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," 2018.

[15] 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.

[16] 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.

[17] 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.

[18] M. Eddington, "Peach fuzzing platform," https://www.peach.tech/products/peach-fuzzer/peach-platform/, [2019-6-1].

[19] 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.

[20] 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.

[21] 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.

[22] gcov, "a test coverage program," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov, [n. d.].

[23] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, 2012.

[24] P. Godefroid, H. Peleg, and R. Singh, "Learn&#38;fuzz: Machine learning for input fuzzing," 2017.

[25] Google, "TriforceAFL," https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/, [2019-6-1].

[26] 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.

[27] 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.

[28] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[29] L. He, Y. Cai, H. Hu, P. Su, Z. Liang, Y. Yang, H. Huang, J. Yan, X. Jia, and D. Feng, "Automatically assessing crashes from heap overflows," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017.

[30] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations," in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE Computer Society, 2012.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, 2018.

[39] 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.

[40] LLVM, "Dataflowsanitizer," https://clang.llvm.org/docs/DataFlowSanitizer.html, [n. d.].

[41] 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.

[42] 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.

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

[44] MSR, "!exploitable," https://archive.codeplex.com/?p=msecdbg, [n. d.].

[45] 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.

[46] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[47] 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.

[48] 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.

[49] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association, 2014.

[50] 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.

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

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

[53] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," 2018.

[54] 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.

[55] 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.

[56] D. Vyukov, "syzkaller," https://github.com/google/syzkaller, [n. d.].

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

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

[66] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, 2018.