Situ LY, Zuo ZQ, Guan L *et al.* Vulnerable region-aware greybox fuzzing. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(5): 1212–1228 Sept. 2021. DOI 10.1007/s11390-021-1196-0

Vulnerable Region-Aware Greybox Fuzzing

Ling-Yun Situ[1,2], *Member, CCF*, Zhi-Qiang Zuo[1,*], *Member, CCF*, Le Guan[3], *Member, ACM, IEEE*
Lin-Zhang Wang[1,*], *Distinguished Member, CCF*, Xuan-Dong Li[1], *Fellow, CCF*
Jin Shi[2], *Member, CCF*, and Peng Liu[4], *Member, ACM, IEEE*

[1] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*
[2] *School of Information Management, Nanjing University, Nanjing 210023, China*
[3] *Department of Computer Science, University of Georgia, Athens, GA 30602, U.S.A.*
[4] *College of Information Sciences and Technology, Pennsylvania State University, State College, PA 16802, U.S.A.*

E-mail: {stly, zqzuo}@nju.edu.cn; leguan@cs.uga.edu; {lzwang, lxd, shijin}@nju.edu.cn; pliu@ist.psu.edu

Received December 3, 2020; accepted May 18, 2021.

Abstract Fuzzing is known to be one of the most effective techniques to uncover security vulnerabilities of large-scale software systems. During fuzzing, it is crucial to distribute the fuzzing resource appropriately so as to achieve the best fuzzing performance under a limited budget. Existing distribution strategies of American Fuzzy Lop (AFL) based greybox fuzzing focus on increasing coverage blindly without considering the metrics of code regions, thus lacking the insight regarding which region is more likely to be vulnerable and deserves more fuzzing resources. We tackle the above drawback by proposing a vulnerable region-aware greybox fuzzing approach. Specifically, we distribute more fuzzing resources towards regions that are more likely to be vulnerable based on four kinds of code metrics. We implemented the approach as an extension to AFL named RegionFuzz. Large-scale experimental evaluations validate the effectiveness and efficiency of RegionFuzz—11 new bugs including three new CVEs are successfully uncovered by RegionFuzz.

Keywords vulnerability detection, greybox fuzzing, code metrics, resource distribution

1 Introduction

Fuzzing [1, 2], as an automatic testing technique, has become one of the most effective approaches to exploring security vulnerabilities in modern large-scale software and systems. It has been widely adopted by industries including Google and Microsoft to improve the reliability and security of their software products. The core idea of fuzzing is to feed a massive number of valid or semi-valid inputs to the target program so as to trigger unintended program behaviors by monitoring the system under testing (SUT).

State-of-the-art fuzzing approaches [3–5] can be classified into three categories according to the usage of internal knowledge of target programs. Blackbox fuzzers [6, 7] are oblivious of the internals of SUT, and thus less effective relatively. Whitebox fuzzers leverage heavy-weight program analysis techniques such as taint analysis [8, 9] or symbolic execution [10, 11] to improve the effectiveness. However, they can hardly scale due to inefficiency of the heavy-weight analysis. Greybox fuzzers such as American Fuzzy Lop (AFL)①, libFuzzer② and honggfuzz③ are in the between. They leverage the

Regular Paper

Invited from Xia Peisu Forum 2020

This work was (partially) supported by the National Key Research and Development Program of China under Grant No. 2017YFA0700604, the National Natural Science Foundation of China under Grant Nos. 62032010 and 61802168, the Leading-Edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001, and the 2021 Double Entrepreneurship Big Data and Theoretical Research Project of Nanjing University.

*Corresponding Author (Lin-Zhang Wang and Zhi-Qiang Zuo are the co-advisers of the first author. They contributed equally to the paper.)

①https://github.com/google/AFL, May 2021.

②https://llvm.org/docs/LibFuzzer.html, May 2021.

③https://github.com/google/honggfuzz, May 2021.

light-weight program analysis and trivial instrumentation to collect runtime information (e.g., coverage and execution time) as feedback, and use the feedback to guide the fuzzing process with the underlying optimization algorithms. Thanks to the tunable flexibility and effectiveness, greybox fuzzers usually achieve better efficacy and performance [5].

Generally speaking, as a random testing technique, a highly effective fuzzing tool is one that can efficiently distribute fuzzing resources. Thus, erroneous program behaviors could be discovered sooner and explored more sufficiently under the limited budget. It is commonly agreed that strategically distributing fuzzing resources could substantially enhance the efficiency of fuzzing [12–14]. Leopard [15] defines its metrics at the function level and computes them based on static analysis alone. [16] guides symbolic execution to less traveled paths based on the dynamic count of paths covered. [17] focuses on the paths' conditions, and performs an evaluation study of concolic testing strategies.

Key Observations. According to the experience of fuzzing large-scale real-world programs using AFL, we summarize some key observations as follows.

1) *Coverage is not always strongly correlated with vulnerability discovery.* Growth in coverage (e.g., branch coverage) is not a good predictor for the increase in the number of unique crashes. Increasing coverage alone may not lead to more vulnerabilities to be discovered [18] in limited resources. The key to vulnerability discovery is whether the code region that vulnerabilities reside is sufficiently explored, rather than just covered. If a vulnerable region is fuzzed more often, more vulnerabilities are likely to be detected under the same budget.

2) *Fuzzing resource distribution based on dynamic metrics alone may cause discrimination for promising vulnerable regions.* Existing fuzzing resource distribution strategies assign the number of test cases for a region based on dynamic metrics alone. As a result, regions that are likely to be vulnerable may be less tested. For example, the dynamic metrics, e.g., the execution time of path, is used to allocate fuzzing resource. The longer the execution time of the path, the less the testing resource assigned. The simple strategy ensures the fuzzing efficiency, but causes the discrimination against the time-consuming paths, which are usually paths with long loops or complex algorithms. Thus, bugs that are usually buried in these time-consuming paths and could cause denial-of-service attacks, would be hardly discovered [19].

Limitation. The above key observations reveal that existing testing resource distribution of AFL-based greybox fuzzing is unaware of the possible vulnerable regions. Existing studies aim to increase the coverage based on specific dynamic metrics without considering any static code metrics, especially vulnerability-related static metrics. Therefore, these strategies lack insight regarding which code region is more likely to be vulnerable and needs more fuzzing resources. It could cause discrimination of fuzzing resources distribution against the promising regions that likely contains vulnerabilities.

Our Work. To tackle the above limitation, we propose a vulnerable region-aware greybox fuzzing approach. More specifically, we schedule the fuzzing resources distribution by considering the static code metrics as well as existing dynamic metrics. We run the test cases and collect values of code metrics as feedback, which are then utilized to distribute fuzzing resources and strengthen fuzzing regions which are more likely to be vulnerable. Four types of vulnerability-related code metrics, namely sensitive, complex, deep and rarely reachable, are considered. We implemented the approach based on AFL and devise a novel fuzzer named RegionFuzz. A comprehensive set of evaluations are conducted to validate the effectiveness of RegionFuzz. Furthermore, RegionFuzz helps us to find 11 new bugs and identifies three new CVEs.

Contributions. The core contributions are summarized as follows.

- *Approach.* We proposed a vulnerable region-aware greybox fuzzing approach, which is able to distribute the fuzzing resources towards those regions that are more likely to be vulnerable.
- *Tool.* We extended AFL by integrating the vulnerable region awareness and developed a new greybox fuzzing tool named RegionFuzz.
- *Vulnerability.* We performed comprehensive evaluations to verify the effectiveness and efficiency of RegionFuzz, which successfully finds 11 unknown bugs and three new CVEs.

The remainder of this paper is organized as follows. Section 2 gives the necessary background of American Fuzz Lop. Section 3 presents our key observations and hypotheses about the resource distribution of AFL-based greybox fuzzing. Section 4 introduces the detailed description of code metric aware fuzzing approach, followed by the implementation of RegionFuzz in Section 5. We conduct experimental evaluations in Section 6, and discuss related work in Section 7. Finally, we conclude the work and present the future plan in Section 8.

2 American Fuzzy Lop

AFL④ is one of the most effective greybox fuzzing tools. It uses sophisticated genetic algorithms to optimize the new test cases generation by employing low-level compile-time instrumentation to collect runtime coverage information. AFL has found numerous significant vulnerabilities in dozens of software projects. Therefore, it has attracted a large and active community which contributes to the project constantly. On top of AFL, a lot of variants such as AFLFast[13], AFLSmart[14] FairFuzz[20], AFLGo[21], CollAFL[22] and Angora[23] have been proposed. They focus on the improvement of certain aspects of AFL respectively.

AFL begins fuzzing the instrumented target program using a set of initial seeds. Then, following an evolutionary fuzzing loop, AFL generates optimized seeds that explore new paths as soon as possible. By this way, it could evolve towards a higher code coverage. In the following, we briefly explain the core internals of AFL for better understanding of the paper.

Instrumentation. AFL instruments the targeted program at compile-time or run-time. The inserted instructions capture basic block transitions, along with coarse branch-taken hit counts. The information is stored in a shared memory buffer, which is accessible by an AFL run-time library. At each branch point, AFL injects code essentially equivalent to Fig.1.

```
cur_location = ⟨COMPILE_TIME_RANDOM⟩;
shared_mem [cur_location^pre_location] ++;
pre_location = cur_location >> 1;
```

Fig.1. Instrumentation of AFL.

The variable *cur_location* identifies the current basic block. It is assigned with a random number at compile time. Buffer *shared_mem* is a 64 KB shared memory region. Every byte that is set in the array marks a hit for a transition represented by a tuple (A, B). Here, tuple (A, B) means a transition from basic block A to basic block B. The shift operation in line 3 preserves the directionality of the transition tuple. Without shifting, the transition from A to B would be indistinguishable from the transition from B to A.

Seed Selection. AFL determines a seed is "favored" if it is the fastest and smallest input for any of the block-to-block transitions it exercises. When selecting seed, "favored" seeds will be chosen with priority, and "unfavored" seeds will be ignored with a random probability.

Energy Assignment. The energy of a seed t refers to the number of new test cases that will be generated from t after applying various mutation operators. In the deterministic stage, AFL determines a seed's energy according to its length. In the havoc stage, AFL firstly determines the basic energy based on its execution time and average execution time. Then it updates total energy based on other attributes such as coverage.

Mutation Selection. Basic mutation operators used in AFL include "flips", "interesting", "arith", "extra" and "splice". In the deterministic stage, these mutation operators are used separately and sequentially to generate new test cases. In this way, AFL can produce compact test cases and small diffs between the non-crashing and crashing inputs. In the havoc stage, AFL mutates the seed by randomly choosing a sequence of mutation operators and applies them to random locations in the seed file.

Feedback Collection. When a new test case is fed to the targeted program and executed by AFL, the feedback information includes code coverage and crash reports will be collected. AFL determines an input to be interesting only if that input has the contribution to code coverage. Intuitively, AFL retains inputs that trigger new block transitions. If the generated input t' crashes the program, it is added to a set of crashing inputs. A crash is considered "unique" if the associated execution paths involve any block-to-block transitions not seen in previously-recorded crashes.

3 Key Observations

We conduct a series of empirical studies of AFL-based greybox fuzzing and obtain some key observations about 1) where more resources should be allocated, and 2) how many resources should be allocated. Furthermore, two hypothesis are proposed about fuzzing resources distribution.

Observation 1. *Coverage Is Not Strongly Correlated with Unique Crashes Discovery.* We empirically study the correlation between paths' number and crashes discovery by running AFL for 24 hours over the well-known benchmarks including LAVA-M[24], Google-fuzzer-testsuit⑤ and so on. The initial seeds for each

④https://github.com/google/AFL, May 2021.

⑤https://github.com/google/fuzzer-test-suite, May 2021.

subject are collected from their testing directory. In addition, AFL's havoc (i.e., using "-d" argument) mode is used to fuzz each subject.

Typically, Fig.2 illustrates the results in terms of path growth and crash growth for each subject program in LAVA-M[24] benchmark, namely base64, md5sum, uniq and who. The x axis indicates the fuzzing time. The y axis on the left represents the number of paths explored, while that on the right indicates the number of crashes found. In Fig.2, the red (full) line and green (dotted) line correspond to the curve of path growth and crash growth with fuzzing time respectively.

Based on the results shown in Fig.2, we could find that the path growths of different fuzzing targets follow the same pattern, while the crash growth varies. As can be seen in Fig.2, the total number of paths increases quickly at the beginning, and the growth rate flattens gradually over time. Finally, the slope of the growth curve is approximately zero, meaning that no new path is triggered. The intuitive reason behind the phenomenon is that fuzzing could be modeled as a weighted coupon collection problem with unknown weight (WCCP)[25]. According to the explanation of the WCCP model, if we assume the total number of unique paths of one program is N, then the probability of finding the i-th new unique path will be $P_i = (N - i + 1)/N$ when $i - 1$ unique paths have been found. Similarly, STADS[26] models fuzzing as the discovery of species, and Böhme Marcel models coverage-based fuzzing as random walking in the Markov chain[13]. They all reveal that the probability of identifying a new unique path becomes lower and lower.

Different from the path growth, there is no universal pattern for the occurrence and growth of crashes. The number growth of crashes found may be relatively slow at the beginning and becomes dramatic at the later stages. The reason behind this phenomenon is that the direct cause of the crash is whether regions that vulnerabilities resided are fully explored[26], rather than covered. Thus, the growth of coverage may improve the probability of triggering more crashes under the "unlimited" fuzzing resource budget, but the coverage growth is not strongly correlated to triggering more bugs within limited resources. Generally speaking, it is true that a higher coverage leads to more crash discovery under the

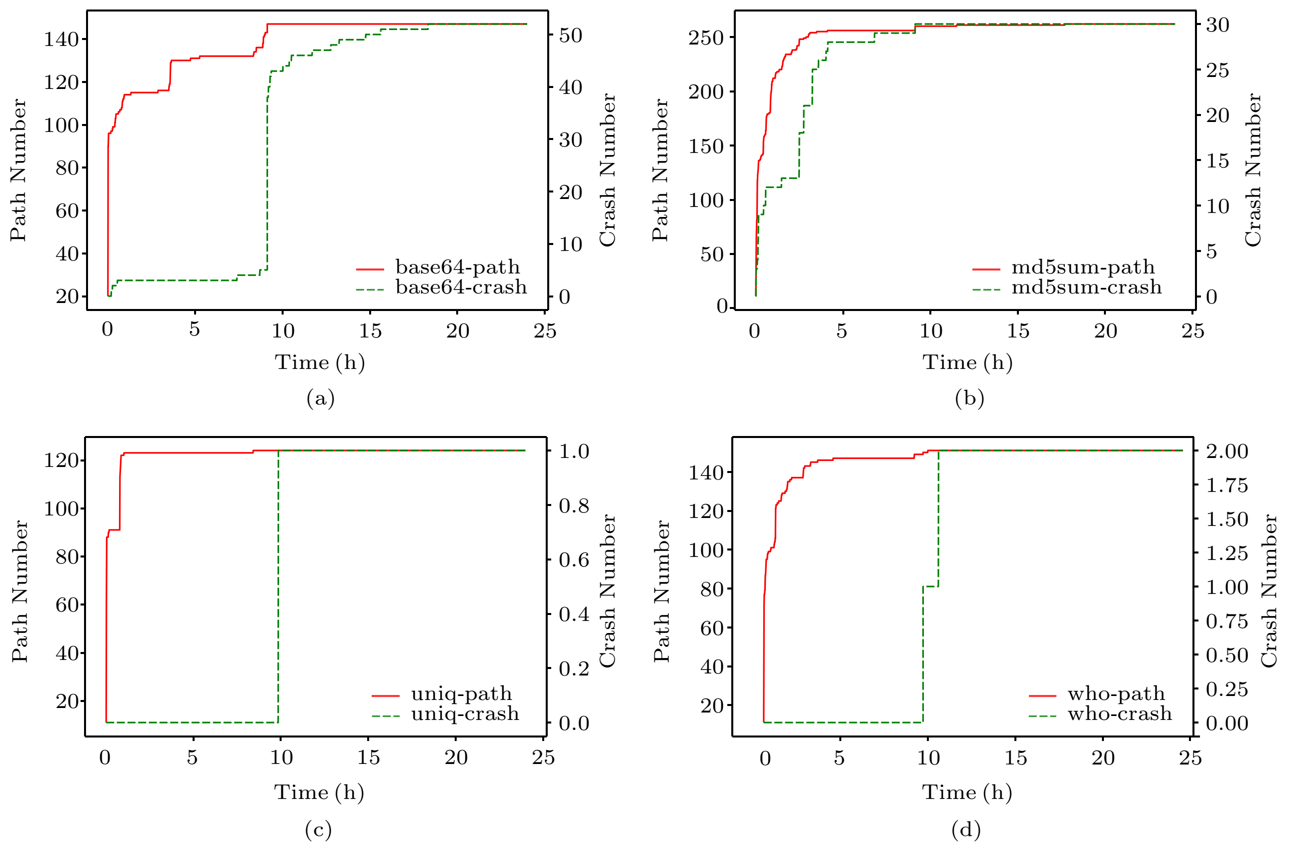


Fig.2. Path and crash growth of LAVA-M. (a) base64. (b) md5sum. (c) uniq. (d) who.

unlimited time and computation resources. However, as shown in Fig.2, it is possible that path coverage grows rapidly while fewer crashes are found. In other words, given the limited fuzzing budget, the coverage growth is not a good predictor of crash growth. Instead, the key should be whether more vulnerability-residing regions are sufficiently explored.

Hypothesis 1. Based on the above observations, we make the following hypothesis for the question, i.e., where more resources should be allocated. If we concentrate more testing resources on regions that are more likely to be vulnerable, then more vulnerabilities could be detected in the given limited fuzzing budget.

Observation 2. *Distributing Resource Based on Dynamic Metrics Alone May Ignore Promising Vulnerable Regions.* Existing energy assignment strategies are based on dynamic metrics alone. For example, the test case with a longer execution time would be assigned a smaller energy. Fig.3 lists the rules used for determining the basis energy of a test case in AFL. *exec_us* represents the execute time of test case q, and *avg_exec_us* represents the average execute time. *perf_score* is the score used for assigning energy. We could obviously read that the maximal basis energy is 30 times as much as the minimal basis.

```
q → exec_us ×0.10 > avg_exec_us → perf_score = 10;
q → exec_us ×0.20 > avg_exec_us → perf_score = 25;
q → exec_us ×0.50 > avg_exec_us → perf_score = 50;
q → exec_us ×0.75 > avg_exec_us → perf_score = 75;
q → exec_us ×2.00 > avg_exec_us → perf_score = 150;
q → exec_us ×3.00 > avg_exec_us → perf_score = 200;
q → exec_us ×4.00 > avg_exec_us → perf_score = 300;
```

Fig.3. Basis energy determination rules.

The dynamic metrics based setting prefers to fast execution paths, which is good for fuzzing efficiency. However, at the same time, it makes the existing energy assignment discriminate against promising vulnerable regions such as the time-consuming paths. They are usually paths with long loops and complex algorithms, and usually bury some vulnerabilities that could cause denial-of-service attacks [19].

We take the code fragment in Fig.4 as an example to show the discrimination. Considering the two paths distinguished by *buf*[0], there are two identical crash bugs (i.e., crash A, crash B) at the end of each path. The existing AFL's energy distribution strategy takes into account the execution time of the test case, and the size of the bitmap. Let the execute time of each instrumentation be one second. The total execute time of path A to trigger crash A is 10 002 s, while the execute time of path B to trigger crash B is 14 s. The bitmap size of path A and path B is 3 and 7 respectively. Based on the computation formula in AFL, the energy score of path A is 7.5 (i.e., 10×0.75) and the energy score of path B is 450 (i.e., 300×1.5). That means the energy of the time-saving path (i.e., path B) is 60 times as much as that of the time-consuming path (i.e., path A). However, crash A is much harder to be detected by AFL than crash B in practice. AFL could trigger crash B in one second, while it must spend more than one hour in triggering crash A. Moreover, if we do not extend the time limit of reporting hang, AFL cannot detect crash A at all due to the long loop within the function. The reason is because that AFLs bitflip quite likely bypasses the single character comparison, whereas it is hard for AFL to bypass the magic bytes comparison. This discrimination makes it difficult to detect vulnerabilities such as crash A behind the long loop.

```
#include "stdio.h"
int main (int argc, char ** argv){
    char buf[8];
    if(read(0, buf, 8) < 1){
      printf("Hum ?\n")
      exit(1);
    }
    if(buf[0] == 'a'){              //Path A
      char * arr;
      for(int i = 0; i < 10000; i++){
         ......
      }
      if(buf[6] = = 'g' && buf[7] = = 'h' ){
        abort();                    //Crash A
      }
    }else if(buf[1] = = 'b'){           //Path B
      if(buf[2] = = 'c'){
        if(buf[3] = = 'd'){
          if(buf[4] = = 'e'){
            if(buf[5] = = 'f'){
              if(buf[6] = = 'g' && buf[7] = = 'h'){
                abort();          //Crash B
              }else printf("error\n");
            }else printf("error\n");
          }else printf("error\n");
        }else printf("error\n");
      }else printf("error\n");
    }
}
```

Fig.4. Code fragment.

Hypothesis 2. Based on the above findings, we make the following hypothesis for the question, i.e., how much resources should be allocated. It may contribute a more reasonable and effective fuzzing energy assignment strategy by combining dynamic and static metrics, especially vulnerability-related static metrics.

4 Approach

We leverage the above observations and hypothesis to improve the fuzzing resources distribution of AFL-based greybox fuzzing in a principle way. We want to make AFL become vulnerable region awareness. Specifically, we automatically identify potential vulnerable regions and schedule the fuzzing energy assignment based on the vulnerability-related metrics at run time. Four kinds of static code metrics are designed, and the energy to each test case is assigned based on the values of these metrics. By using the light-weight static analysis, we extract code metrics including sensitive degree, complexity, depth and rare reachable degree from the target program. Then, we instrument the values of each code metric into the target program. At run time, the reward of a test case is calculated by summing up the value of code metric along its execution path. The reward is used to calculate the energy for the test case that triggers new coverage.

4.1 Code Metrics

Although there is no study claiming a specific code metric that could perfectly assess vulnerable code regions, some heuristics do exist[15–17]. Typically, the sensitive regions, complex regions, deep regions, and rarely reachable regions are more likely to contain vulnerabilities due to the code complexity and/or the lack of adequate testing.

We devise four types of code metrics (i.e., sensitive degree, complex degree, deep degree and rare-reach degree), and facilitate them to identify potential vulnerable regions and assign fuzzing energy effectively.

Sensitive Degree. The common memory corruption vulnerabilities (e.g., use-after-free, buffer overflow, format string) in C/C++ programs are usually caused by memory and string-related operations. If a code region contains more memory and string related sensitive functions such as strcpy, memcpy and so on, it is more likely to include memory-related vulnerabilities[27]. Based on the above intuition, the sensitive metric is designed to measure the sensitiveness of the code regions (i.e., basic blocks) as (1).

$$Degree_{sen}(BB) = MemOP(BB) + StrOP(BB), \quad (1)$$

where $MemOP(BB)$ and $StrOP(BB)$ represent the total numbers of memory and string-related instructions within the basic block BB, respectively.

Complex Degree. If a code region involves more complex logic, the programmer is prone to introducing vulnerabilities during coding[19]. Based on the above intuition, we propose a complexity metric to measure the complexity of code regions. Different from the complexity measurement for a function, what we measure is the basic block. Thus, we compute the sum of entry degree and the number of call instruments of a basic block as indicators to the complexity of a basic block, leading to (2) to calculate the complex degree.

$$Degree_{com}(BB) = Pred(BB) + CallInst(BB), \quad (2)$$

where $Pred(BB)$ represents the total number of a basic block's predecessors, which is used to represent its connection complexity. $CallInst(BB)$ represents the total number of call instruments in the basic block BB, which is used to represent its logic complexity. In general, if logic is implemented with more function call instruments, the complexity is higher.

Deep Degree. Deep code regions hide up bugs due to the lake of sufficient testing. Fuzzing deep regions achieves more chances to bug discovery[28]. Here, we define the depth metric, which is utilized to guide fuzzing into deep regions. For a given basic block BB, we compute the depth degree of BB as (3).

$$Degree_{deep}(BB) = \frac{P(BB).size}{\sum_{p_i \in P(BB)} \frac{1}{length(p_i)}}, \quad (3)$$

where $P(BB)$ denotes all the possible paths from the entry block to block BB within the function scope. $length(p_i)$ measures the length of path p_i, which is the total number of blocks along path p_i. Note that for a loop path, we only count the blocks within the loop once.

Taking Fig.5 as an example, to calculate the depth degree of basic block G, CFG is firstly traversed using the deep first search (DFS) algorithm to obtain all the paths from entry block A to G. They are $p_1 = A \to B \to D \to G$, $p_2 = A \to B \to D \to F \to G$, $p_3 = A \to C \to D \to G$, $p_4 = A \to C \to D \to F \to G$. $p_5 = A \to C \to E \to F \to G$. The lengthes of the above paths are 3, 4, 3, 4, 4 respectively. Then, the depth degree of basic block G is computed as (3), and the result is $1/(1/3 + 1/4 + 1/3 + 1/4 + 1/4) = 12/17$. Note that the depth is computed intra-procedurally without considering the distance crossing functions.

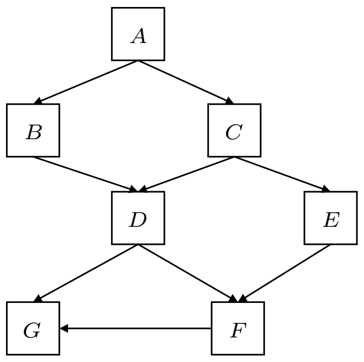


Fig.5. Abstract CFG.

Rare-Reach Degree. It is claimed that less traversed regions have higher chance to be vulnerable[16]. If a region has a lower probability of being reached, vulnerabilities buried in these regions are harder to be detected. Based on this, the rare-reach metric is devised to indicate the rarely reachable degree of a region.

For a given basic block BB, we first compute the probability $RPro(BB, p)$ of each path p reaching BB as shown as (4).

$$RPro(BB, p) = \frac{1}{\prod_{j=0}^{length(p)-1} TPro(B_j, B_{j+1})}, \quad (4)$$

where p indicates a path reaching BB. $TPro(B_j, B_{j+1}) = 1/successors(B_j)$, $length(p)$ represents the length of path p, and $successors(B_j)$ represents the number of successor basic blocks of the basic block B_j. We assume the probability of transiting from one block to any of its successors is equal. Note that this assumption is not true in reality. But we cannot calculate the transition probability statically. Thus, different from other heavy-weight approaches to computing a more accurate probability dynamically[17], we adopt this assumption due to a trade-off between accuracy and performance. Having the reachable probability for all the paths reaching BB, the rarely reachable degree of BB is computed as (5):

$$Degree_{rare}(BB) = \frac{P(BB).size}{\sum_{p_i \in P(BB)} RPro(BB, p_i)}. \quad (5)$$

For instance, we compute the reachability degree of block D in Fig.5. Firstly, we get the path list $P(D)$ containing all the paths from the entry block A to D. In this example, $P(D)$ consists of two paths $p_1 = A \to B \to D$ and $p_2 = A \to C \to D$. Based on the successor information, we calculate the reachable probability for each path: $RPro(D, p_1) = 1/(1/2 \times 1)$ and $RPro(D, p_2) = 1/(1/2 \times 1/2)$. Finally, the reachability degree $Degree_{rare}(D) = 2/(2 + 4) = 1/3$.

4.2 Energy Assignment

A test case is assigned energy according to the weights of its execution path's code metric. The path with a higher metric weight will be assigned with more fuzzing energy. As a result, these paths that are more sensitive, more complex, deeper or more rare reachable will be fuzzed more and fully explored.

For a given test case t, we calculate its reward $Reward(t)$ at run time. The reward is defined as (6), where $p(t)$ denotes the list of basic blocks executed by the test case t, BB_i represents the i-th basic block, n is the total number of basic blocks within $p(t)$, and $Degree_M(BB_i)$ is the reward of a specific block BB_i for metric M. M represents one of four types of code metrics, i.e., sensitive degree, complex degree, deep degree and rare-reach degree.

$$Reward(t) = \frac{\sum_{BB_i \in p(t)} Degree_M(BB_i)}{p(t).size}. \quad (6)$$

For example, if a test case t leads to a path $p = A \to B \to D \to G$, its reward $Reward(t)$ can be calculated by $(Degree_M(A) + Degree_M(B) + Degree_M(D) + Degree_M(G))/4$.

During fuzzing, we maintain the average reward $AvgReward$ among all test cases. An energy distribution factor F for each test case t is computed based on a seed's reward (i.e., $Reward(t)$) and the overall average reward (i.e., $AvgReward$) as (7).

$$F(t) = \frac{Reward(t)}{AvgReward}. \quad (7)$$

The larger the F value of t is, the more the energy is assigned for t. More specifically, an exponential energy assignment formula in the following is used to assign energy based on factor F. Let $E_{afl}(t)$ be the energy assigned by AFL's energy assignment criteria (e.g., execution time, coverage) for input t, and $E(t)$ be the energy assigned by the region-aware distribution mechanism. $E(t)$ can be computed as (8):

$$E(t) = E_{afl}(t) \times 2^{10 \times F(t)}. \quad (8)$$

5 Implementation

We incorporate the aforementioned improvements into the latest afl-2.52b[6], and develope a new fuzzing tool named RegionFuzz .

[6] https://lcamtuf.coredump.cx/afl/, Sept. 2021.

An overview of RegionFuzz is illustrated in Fig.6. We modify the AFL's instrumentation, attributes computation, seed selection and energy assignment parts, which are specified as the green parts of Fig.6. Firstly, code metrics including a sensitive degree, complexity, depth, and rare-reach degree are extracted from the target program. We invoke light-weight intra-procedural static analysis for this purpose. Then we incorporate the weights of basic blocks regarding these semantic metrics through compile-time instrumentation. As a result, the target program will feedback the rewards of executed paths. These path semantic rewards are later used for distributing fuzzing energy. RegionFuzz prefers test cases gaining higher rewards and assigns more energy to them. In this way, code regions that are more likely to contain a vulnerability based on code metrics will be allocated more fuzzing resources. The details of implementation are described as follows.

Code Metric Extraction. We implement LLVM passes to extract weight for code metrics. These metrics include sensitive degree, complexity, depth, and rare-reach degree. These passes are invoked by the compiler *afl-clang-preprocess* as long as they are enabled. Specifically, individual passes are enabled by corresponding environment variables. The weights of basic blocks are stored in a text file named weight_file after processing.

Compile-Time Instrumentation. The weight_file stores the identification of each basic block and its weight value for each metric. It is used by the compiler to instrument the weight values into the target binary. More specifically, an extended trampoline is injected to each basic block. The trampoline is a piece of assembly code that is executed after the jump instruction. It keeps track of the coverage information in form of control-flow edges. An edge is represented by a byte in a shared memory of 64 KB. On 64-bit architectures, we use additional 16 bytes for each edge to record the reward feedback. Eight bytes are used to accumulate weight values, while the other eight bytes are used to record the number of executed basic blocks. The instrumentation is implemented as an extension to AFL LLVM pass.

Energy Assignment. RegionFuzz fuzzes the instrumented binary with the proposed energy distribution strategy. In particular, it first selects seeds and assigns energy based on run-time rewards of test cases. The current test case's reward is then computed by dividing accumulated basic block weight by the number of exercised basic blocks. Note that this operation is performed for each different semantic metric. RegionFuzz selects test cases that gain higher rewards and assigns more energy for them based on the seed's reward factor.

6 Evaluation

We conducted comprehensive evaluations to validate the effectiveness of vulnerable region-aware greybox fuzzing approach, and show the performance of RegionFuzz by comparing it with multiple state-of-the-art greybox fuzzers.

6.1 Experimental Setup

Research Questions. The experimental evaluations were performed to answer the following research ques-

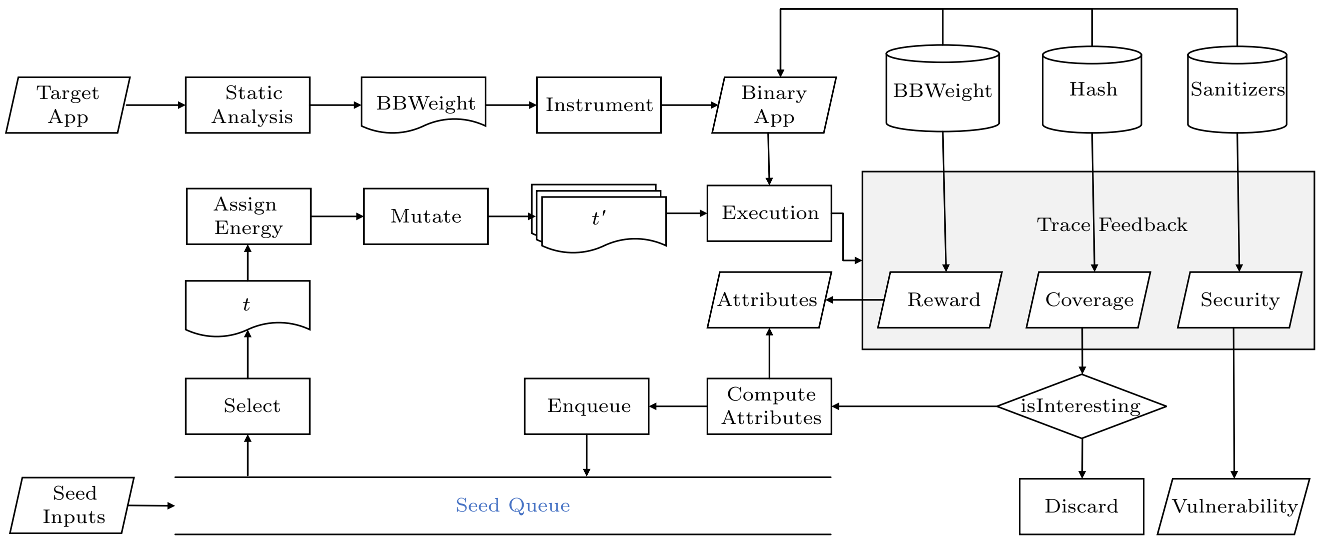


Fig.6. Overview of vulnerable region-awareness greybox fuzzing.

tions.

• *RQ*1. How is the effectiveness of RegionFuzz's code metrics awareness? (Subsection 6.2)

• *RQ*2. How is the comparison of RegionFuzz and related AFL-based greybox fuzzing tools? (Subsection 6.3)

• *RQ*3. How is the ability of RegionFuzz to detect vulnerabilities in practice? (Subsection 6.4)

Measurement Metrics. The experimental evaluations were designed to validate the effectiveness of code metrics awareness and the performance of RegionFuzz comparing against state-of-the-art fuzzers. The following measurement metrics were calculated to answer the above research questions:

• the time to trigger the first crash (i.e., first-crash);

• the total number of unique crashes found (i,e., #crash);

• the total number of paths explored under a given time budget (i.e., #path).

For the sake of robust assessments claimed by [29], we ran each of the fuzzers 10 times and reported the average values.

Evaluation Subjects. A widely-used benchmark (i.e., LAVA-M) and some popular open source projects from google fuzzer testsuit[7] were selected as the fuzzing subjects in Table 1. These subjects are known to have vulnerabilities, and hence form a ground-truth for evaluating fuzzing tools.

More specifically, LAVA-M consists of four buggy versions of Linux utilities, i.e., base64, md5sum, uniq and who. It was generated by automatically injecting known vulnerabilities into the source code [24]. It has been commonly applied as fuzzing benchmarks in a lot of research work [13, 20, 22, 23, 28].

In addition, some open source projects were selected for evaluation based on the following criteria: popularity in the community, development activeness, and diversity of categories. We selected several old versions, namely libxml2-2.9.2, libtiff-3.7.0, bison-3.0.4, clfow-1.5 and libjpeg-tubo-1.2.0, which contain the known vulnerabilities and the ground-truth information for evaluation comparison.

Evaluation Tools. To understand the effectiveness of our energy distribution strategy, we compared RegionFuzz against several state-of-the-art AFL-based greybox fuzzers whose source code is publicly accessible.

• *AFL* is the official latest AFL implementation (i.e., AFL-2.52b);

• *AFLFast* is an AFL variant spending more energy on low-frequency paths [13];

• *FairFuzz* is an AFL variant spending more energy on low-frequency branches [20];

• *TortoiseFuzz* is an AFL-based greybox fuzzer based on coverage accounting [30];

• *RegionFuzz* is our proposed code metric-aware fuzzer.

Note that we did not make comparison with AFLGo [21] or Hawkeye [31] as they have different prerequisites and goals. AFLGo and Hawkeye are two concrete target-directed fuzzing tools, which require prior knowledge such as pre-provided target locations and aim to reach them quickly. Instead, RegionFuzz is a heuristic-based randomly fuzzing tool and aims to sufficiently explore vulnerable regions within a limited budget. It identifies possible vulnerable regions automatically based on code metrics and distributes more energy for them based on metrics values, in order to strengthen fuzzing these regions.

Experimental Infrastructure. All the fuzzing tools in our experiments run on a virtual machine with eight 2 GHz Intel CPU cores and 8 GB RAM on Ubuntu 16.04.

Fuzzing Configuration. The initial seeds for each project were collected from their testing directory. In addition, we used AFL's havoc mode to fuzz each subject (i.e., using "-d" argument).

Table 1. Benchmark for Evaluation

Project	Size	Function Description
base64	160.0 KB	Encode or decode file
md5sum	221.0 KB	Print or check MD5 (128-bit) checksums
uniq	186.0 KB	Filter adjacent matching lines from input
who	167.0 KB	Print information about users who are currently logged in
libxml2-2.9.2	195.0 KB	Parse the XML files
libtiff-3.7.0	219.0 KB	Translate tiff file into pdf file
bision-3.0.4	1.8 MB	Generate a deterministic LR or a generalized LR
cflow-1.5	609.0 KB	Generate a program flow graph
libjpeg-tubo-1.2.0	143.0 KB	Switch format of picture

[7]https://github.com/google/fuzzer-test-suite, May 2021.

6.2 Effectiveness of Code Metric Awareness

In order to evaluate the effectiveness of the code metric awareness, we implemented four fuzzers based on the official afl-2.52b codebase, each with one code metric enabled. We named them as RegionFuzz-sen (sensitive degree), RegionFuzz-com (complex degree), RegionFuzz-deep (deep degree) and RegionFuzz-rare (rare-reach degree) respectively. We chose the traditional AFL as a baseline and executed the four code metrics-aware greybox fuzzers on the benchmarks. Note that each subject was fuzzed 10 times, each for 24 hours, in order to reduce the randomness. For each fuzzer, the three columns report the time in minutes to trigger the first crash (i.e., first-crash), the number of unique crashes found (i.e., #crash), and the total number of paths explored (i.e., #path), respectively. The Vargha-Delaney statistic is a non-parametric measure of an effect size [32] and is also the recommended standard measure for the evaluation of randomized algorithms [33]. Given a performance measure metric such as first-crash, #crash or #path, m measures of RegionFuzz and n measures of AFL, the statistics measure the probability that running RegionFuzz yields better than running AFL. We used Man-Whitney U test to measure the statistical significance of performance gain.

Time to Trigger the First Crash. The time to trigger the first crash is an important factor to evaluate a fuzzing tool with guidance features. The results are illustrated in Table 2, the improvement represents the proportion of shortened time of finding the first crash compared with AFL, and the p value represents the statistical significance of performance gain. As can be seen in Table 2, most of the four code metrics-aware fuzzers outperform AFL in finding the first crash for most cases, and the statistical significance is obvious. The advance ratio for detecting the first crash are 38.49%, 30.41%, 20.12%, and 31.32% for the four kinds of code metric based fuzzing energy distribution strategies, respectively. The p values are 0.21, 0.35, 0.28 and 0.25 respectively. Generally, the sensitive degree guided fuzzing performs the best in triggering the first crash over the selected benchmarks. This is reasonable as sensitive regions bury vulnerabilities due to the usage of memory- and string-related operations. With the sensitive metrics considered, strengthening fuzzing sensitive paths can quickly discover certain bugs. In specific cases, the sensitive degree-directed strategy performs extremely well. For bison-3.0.4, even 95.43% improvement is achieved. It turns out that the code regions involving more memory- and string-related functions have a higher probability to contain memory corruption bugs. Note that all fuzzers have the same results for libtiff-3.7.0. This is because the crash of libtiff-3.7.0 is located in the entry function, which is readily to be triggered. All the fuzzers can reach it quickly without significant time difference.

Unique Crashes. The total number of unique crashes is another key factor to show the effectiveness of code metrics-aware fuzzing. Although the crashes with the same root cause may result in multiple unique crashes, and some crashes are even not exploitable by outside attackers, identifying more unique crashes indicates better fuzzing ability and higher chances to find real vulnerabilities [31]. The results are illustrated in Table 3, the improvement represents the proportion of the improved number of detected crashes compared with AFL, and the p value represents the statistical significance of performance gain. As can be seen in Table 3 apparently, four path semantic-aware fuzzers reported more unique crashes in most cases than the original AFL. The average improvements are 14.77%, 24.15%, 18.56%, 27.35% respectively. The p values are 0.36, 0.29, 0.33 and 0.23 respectively, which show that the performance gains are obvious. For some specific case

Table 2. Time to Trigger the First Crash of Four Code Metrics Based Fuzzing

Project	AFL	RegionFuzz-sen	RegionFuzz-com	RegionFuzz-deep	RegionFuzz-rare
base64	23.15	15.13	14.16	10.68	19.40
md5sum	3.95	3.23	2.55	1.86	2.81
uniq	1 072.65	751.29	996.78	162.93	659.13
who	937.31	879.33	1 215.32	793.22	1 073.71
libxml2-2.9.2	834.61	209.30	644.45	342.65	233.08
libtiff-3.7.0	0.08	0.08	0.08	0.08	0.08
bision-3.0.4	52.14	2.38	4.88	21.70	14.53
cflow-1.5	22.63	7.07	10.18	34.37	9.42
libjpeg-tubo-1.2.0	117.25	95.63	54.26	244.25	104.9
Improvement	–	+38.49%	+30.41%	+20.12%	+31.32%
p value	–	0.21	0.35	0.28	0.25

Table 3. Total Crashes (#crash) of Four Code Metrics Based Fuzzing

Project	AFL	RegionFuzz-sen	RegionFuzz-com	RegionFuzz-deep	RegionFuzz-rare
base64	53	69	68	61	88
md5sum	32	39	28	43	37
uniq	1	1	1	1	1
who	2	2	2	2	2
libxml2-2.9.2	12	17	8	28	45
libtiff-3.7.0	52	63	61	71	78
bision-3.0.4	161	190	169	212	193
cflow-1.5	166	176	240	160	175
libjpeg-tubo-1.2.0	22	18	45	16	19
Improvement	–	+14.76%	+24.15%	+18.56%	+27.35%
p value	–	0.36	0.29	0.33	0.23

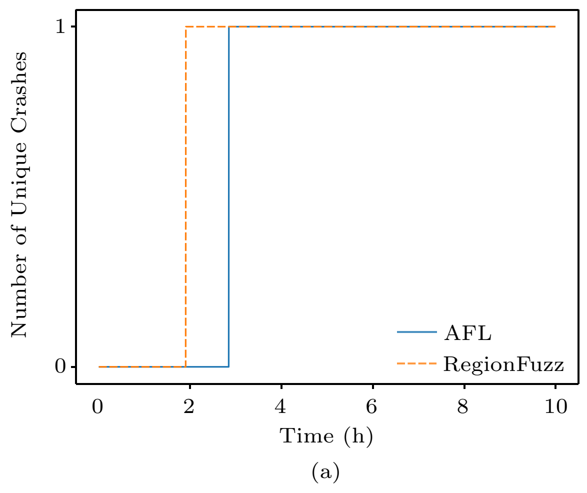
such as libxml2-2.9.2, the improvement by RegionFuzz-rare reaches 216.66%. The results provide the strong evidence to the effectiveness of code metrics-aware energy distribution. The results also indicate that concentrating more fuzzing energy on more vulnerable paths triggers more crashes than maximizing code coverage.

Crashes Growth over Time. Furthermore, we compared RegionFuzz with AFL in terms of the crash growth over time. We randomly selected two target projects, namely LAVAM-M-uniq and ImageMagic-7.0.8, where uniq belongs to the LAVA benchmark and ImageMagic-7.0.8 is a real-world program. Fig.7 plots the detailed results.

Apparently, RegionFuzz shows a larger correlation between the crashes triggered and paths covered than AFL. It indicates that RegionFuzz performs better than AFL in both triggering the first crash and the total number of crashes.

Total Paths. The total number of paths explored within a limited time budget is another common factor to measure fuzzing ability. The results are illustrated in Table 4, the improvement represents the proportion of improved number of detected crashes compared with AFL, and the p value represents the statistical significance of performance gain. As can be seen in Table 4, our four code metrics based fuzzers achieve 3.74%, 4.27%, 2.22% and 17.26% improvements, respectively. The p values are 0.48, 0.47, 0.48, and 0.33 respectively. The rarely reachable regions-guided strategy is the most effective in growing paths. Note that RegionFuzz-deep guides fuzzing to explore the deep code region, which may cause the fuzzer to be stuck in a deep loop area. That is why RegionFuzz-deep has poor performance for cflow-1.5 and libjpeg-tubo-1.2.0 in terms of the total crashes and paths.

Summary. Based on the above observations, we could answer RQ1 that code metric awareness is effective to improving the fuzzing efficiency.



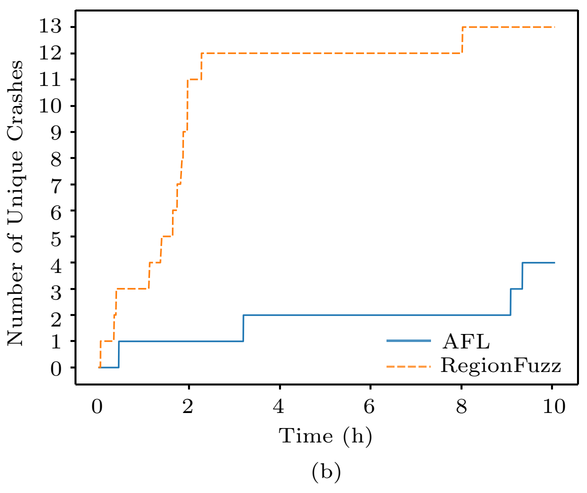


Fig.7. Growth of unique crashed detected by AFL and RegionFuzz. (a) Growth of unique crashes in LAVAM-M-uniq. (b) Growth of unique crashes in ImageMagick-7.0.8.

Table 4. Total Paths (#path) of Four Code Metrics Based Fuzzing

Project	AFL	RegionFuzz-sen	RegionFuzz-com	RegionFuzz-deep	RegionFuzz-rare
base64	155	131	148	151	288
md5sum	370	379	361	386	412
uniq	126	126	128	132	144
who	202	218	187	224	216
libxml2-2.9.2	6 080	6 415	6 311	6 528	7 331
libtiff-3.7.0	469	541	520	521	571
bision-3.0.4	4 370	4 677	4 529	4 503	5 409
cflow-1.5	1 634	1 624	1 665	1 621	1 782
libjpeg-tubo-1.2.0	2 908	2 813	3 161	2 610	2 978
Improvement	–	+3.74%	+4.27%	+2.22%	+17.26%
p value	–	0.48	0.47	0.48	0.33

6.3 Comparison with Related Tools

We compared RegionFuzz with AFL, AFLFast, FairFuzz, and TortoiseFuzz on selected benchmarks. Note that we used the rear-reach code metric in the following evaluation, because it performs better in both the path growth and the crash discovery. Table 5 and Table 6 give the experimental results. Note that the improvement represents the proportion of the shortened time of detecting the first crash, the improved number of detected crashes, and the improved number of covered paths compared with AFL, and the *p* value represents the statistical significance.

Time to Trigger the First Crash. As shown in Table 5 and Table 6, RegionFuzz is good at quickly reaching the first crash. It saves time triggering the first crash by 31.32% compared with the latest AFL, the *p* value is 0.25, and it also outperforms AFLFast, FairFuzz, and TortoiseFuzz in general.

Unique Crashes. Compared with AFL and AFLFast, RegionFuzz detects more crashes within 24 hours on the benchmarks. The improvements on the number of crashes discovered by AFLFast and RegionFuzz are 11.38% and 27.35%, respectively. In our evaluation, FairFuzz performs poorly (i.e., −11.98%). None of crashes is reported by FairFuzz for base64, uniq, and who. TortoiseFuzz also performs worse than AFL in general.

Total Paths. As for the total paths explored, RegionFuzz outperforms all the other four fuzzers. It accomplishes 17.26% improvement on average, and the *p* value is 0.33. The average improvements by AFLFast, FairFuzz, and TortoiseFuzz are 7.69%, −9.15%, and +0.8%, respectively. And the *p* values of AFLFast, FairFuzz and TortoiseFuzz are 0.38, 0.33, and 0.19 respectively.

Summary. Based on the above observations, we answer RQ2 that RegionFuzz achieves better performance than existing AFL-based greybox fuzzing tools.

6.4 Vulnerability Discovery

With the advantage of vulnerable region awareness, RegionFuzz has helped us to find 11 unknown bugs as shown in Table 7.

Furthermore, three new CVEs were assigned as

Table 5. Comparison Results of AFL, AFLFast and FairFuzz

Project	AFL			AFLFast			FairFuzz		
	First-Crash	#crash	#path	First-Crash	#crash	#path	First-Crash	#crash	#path
base64	23.15	53	155	59.46	52	139	NA	0	116
md5sum	3.95	32	370	2.13	37	378	5.81	30	301
uniq	1 072.65	1	126	901.01	1	132	NA	0	134
who	937.31	2	202	890.33	2	209	NA	0	179
libxml2-2.9.2	534.61	12	6 080	560.23	17	6 402	826.70	22	6 410
libtiff-3.7.0	0.08	52	469	0.08	52	512	0.08	62	490
bison-3.0.4	52.14	161	4 370	42.23	208	5 016	5.68	119	4 014
cflow-1.5	22.65	166	1 634	5.81	177	1 694	33.52	204	1 497
libjpeg-tubo-1.2.0	117.25	22	2 908	58.5	12	3 087	768.03	4	1 681
Improvement	–	–	–	+0.54%	+11.38%	+7.69%	−102.55%	−11.98%	−9.15%
p value	–	–	–	0.49	0.45	0.38	0.23	0.21	0.33

Table 6. Comparison Results of AFL, TortoiseFuzz and RegionFuzz

Project	AFL			TortoiseFuzz			RegionFuzz-Rare		
	First-Crash	#crash	#path	First-Crash	#crash	#path	First-Crash	#crash	#path
base64	23.15	53	155	9.63	56	115	19.40	88	288
md5sum	3.95	32	370	1.61	15	340	2.81	37	412
uniq	1 072.65	1	126	962.93	1	93	659.13	1	144
who	937.31	2	202	120.46	1	219	1 073.71	2	216
libxml2-2.9.2	534.61	12	6 080	378.12	39	7 019	233.08	45	7 331
libtiff-3.7.0	0.08	52	469	0.08	59	522	0.08	78	571
bison-3.0.4	52.14	161	4 370	4.62	150	3 994	14.53	193	5 409
cflow-1.5	22.65	166	1 634	52.61	157	1 242	9.42	175	1 782
libjpeg-tubo-1.2.0	117.25	22	2 908	79.40	16	2 980	104.90	19	2 978
Improvement	–	–	–	+26.11%	−1.39%	+0.8%	+31.32%	+27.35%	+17.26%
p value	–	–	–	0.18	0.36	0.19	0.25	0.23	0.33

Table 7. Bugs Found by RegionFuzz

Bug ID	Project	Bug Type	URL
Bug-2018-08140	bison-3.0.4	Assert Abortion	https://github.com/Distrotech/bison/issues/1
Bug-2018-08141	jasper-2.0.14	Assert Abortion	https://github.com/jasper-software/jasper/issues/183
Bug-2018-08142	jasper-2.0.14	Assert Abortion	https://github.com/jasper-software/jasper/issues/183
Bug-2018-0822	nasm-2.14rc15	Null Pointer Deference	https://bugzilla.nasm.us/show_bug.cgi?id=3392507
Bug-2018-0906	nasm-2.14rc15	Stack Overflow	https://bugzilla.nasm.us/show_bug.cgi?id=3392514
Bug-2018-0910	nasm-2.14rc15	Integer Overflow	https://github.com/cyrillos/nasm/issues/4
Bug-2018-0912	nasm-2.14rc15	Stack Overflow	https://github.com/cyrillos/nasm/issues/5
Bug-2017-1109	jabberd2-2.6.1	Buffer Overflow	https://github.com/jabberd2/jabberd2/issues/159
Bug-2017-1110	jabberd2-2.6.1	Buffer Overflow	https://github.com/jabberd2/jabberd2/issues/160
Bug-2018-0822	libtasn1-4.13	Memory Consumption	https://gitlab.com/gnutls/libtasn1/-/issues/4
Bug-2018-0518	Zephyr-1.13.0	Null Pointer Deference	https://github.com/zephyrproject-rtos/zephyr/issues/7638

follows: 1) CVE-2018-1000654, Denial of Service, libtasn1-4.13; 2) CVE-2018-1000667, Null Pointer Reference, nasm-2.14rc15; 3) CVE-2018-1000886, Stack Overflow, nasm-2.14rc15.

In the following, we use two vulnerabilities (i.e., CVE-2018-1000654, CVE-2018-1000667) as case studies to illustrate the effectiveness of the metrics-based guidance.

Case 1: *CVE-2018-1000654.* This is a vulnerability found by RegionFuzz in libtasn1-4.13, which is able to cause a denial of service attacks. More specifically, the CPU usage will reach 100% when running asn1Paser against the proof-of-concept test case. The detailed crash dump is shown in Fig.8.

The core dump of the crash information indicates that there are a lot of string-related operations performed before uncovering the bug. Thus, taking into account the number of string- and memory-related operations as sensitive metrics, RegionFuzz will spend more resources on these regions and paths with a higher sensitive degree, i.e., a larger number of string and memory related operations. That is why RegionFuzz is able to identify the vulnerability more quickly (i.e., 3 hours earlier) than AFL itself in practice. For more detailed information, please refer to the link[8].

```
stly@ubuntu:~/stly/RegionFuzz/libtasn1-4.13/$ ./as-
    n1Parser-c crashes/id\:000000, rep\:1234:
crashes/id:000000,rep:2:23: Warning: UniversalString.
crashes/id:000000,rep:2:56: Warning: VisibleString.
crashes/id:000000,rep:2:58: Warning: NumericString.
crashes/id:000000,rep:2:60: Warning: IA5Stringe.
crashes/id:000000,rep:2:62: Warning: TeletexString
crashes/id:000000,rep:2:64: Warning: PrintableString
crashes/id:000000,rep:2:66: Warning: UniversalString
crashes/id:000000,rep:2:92: Warning: VisibleString
.....
crashes/id:000000,rep:2:167: Warning: IA5String
crashes/id:000000,rep:2:169: Warning: TeletexString
crashes/id:000000,rep:2:171: Warning: PrintableString
Killed
```

Fig.8. Crash dump of CVE-2018-1000654.

[8]https://nvd.nist.gov/vuln/detail/CVE-2018-1000654, May 2021.

Case 2: *CVE-2018-1000667.* This is a null pointer dereference vulnerability found by RegionFuzz in nasm-2.14rc15. This vulnerability could crash the nasm when it processes a crafted file. The entry function is *assemble_file*(*inname, depend_ptr*), and the crash point is located in the function *do_directive*(*Token * tline*) of preproc.c file. The detailed core dump of the crash is illustrated in Fig.9, and the vulnerable function is shown in the Fig.10.

do_directive is a complex function with 1 464 lines of code. *tt*→*text* is set to NULL. When calling *parse_size* with NULL, the crash happens due to the dereference of a null pointer in function *parse_size*. RegionFuzz takes the function's complexity into consideration when distributing the fuzzing resources. As a complex function, *do_directive* has the priority to be fuzzed. As such, RegionFuzz is more likely to identify the example vulnerability than AFL itself. Please refer to the link[9] for more details.

Summary. Based on above observations, we answer RQ3 that our RegionFuzz could effectively identify new bugs and unknown vulnerabilities based on the metrics.

```
stly@ubuntu:~/stly/RegionFuzz/nasm-2.14rc15$ ./nasm
    -felf64 ./crashes/id\:000000\,rep\:16
./crashes/id:000000,rep:16:4:
error: unknown preprocessor directive '%ar'
./crashes/id:000000,rep:16:4:
error: '%$locazeBflat': context stack is empty
./crashes/id:000000,rep:16:4:
error: '%$locazeBflat': context stack is empty
./crashes/id:000000,rep:16:4:
error: '%$locazeBflat': context stack is empty
./crashes/id:000000,rep:16:4:
error: label or instruction expected at start line
./crashes/id:000000,rep:16:5:
error: '%$localsize': context stack is empty
......
./crashes/id:000000,rep:16:5:
error: expression syntax error
ASAN:DEADLYSIGNAL
------------------------------------
SUMMARY: AddressSanitizer:
SEGV (/stly/RegionFuzz/nasm-2.14rc15/nasm+0x44cba8)
==25104==ABORTING
```

Fig.9. Crash dump of CVE-2018-1000667.

```
static int do_directive (Token *tline, char **output)
{
  ...
  switch (i) {
  case PP_INVALID:
     nasm_error(ERR_NONFATAL,
       "unknown preprocessor directive '%s'",
        tline->text);
     return NO_DIRECTIVE_FOUND;

  case PP_ARG:
     offset = ArgOffset;
     do {
        ...
       /*Allow macro expansion of type parameter*/
       tt = tokenize(tline->text);
       tt = expand_smacro(tt);
       size = parse_size(tt->text);
       if (!size) {
          nasm_error(ERR_NONFATAL,
            "Invalid size type for
            '%% arg' missing directive");
          free_tlist(tt);
          free_tlist (origline);
          return DIRECTIVE_FOUND;
       }
       free_tlist (tt);
      ...
    }
```

Fig.10. expand_mmac_params.

7 Related Work and Discussion

We review related work that improves AFL's performance in terms of effectiveness and efficiency.

Improving Feedback. Steelix[34] instruments AFL to collect the progress information of magic bytes comparison, and help understand how to craft mutations to bypass magic bytes comparisons. CollAFL[22] demonstrates that the inaccuracy of feedback (i.e., hash collision issue) in AFL would limit the effectiveness of new path discovery. The authors[22] designed an algorithm to resolve the hash collision problem and improve the edge coverage accuracy with a low-overhead instrumentation mechanism. We propose four code metrics as feedback to identify potential vulnerable regions at runtime.

Improving Sensitivity to Security Violation. Fuzzers usually rely on program crashes as an indicator of vulnerabilities. Researchers proposed various solu-

[9]https://nvd.nist.gov/vuln/detail/CVE-2018-1000667, May 2021.

tions to make the program more sensitive to various security violations such as AddressSanitizer[35] and MemorySanitizer[36]. They instrument the program to trace memory usage and invoke error-handling code whenever an unexpected memory access is detected. There are many other sanitizers available like DataFlowSanitizer, ThreadSanitizer[37] and so on. All the sanitizers are utilized only when a vulnerable region is already reached. They are not able to identify potential vulnerable regions in advance. In addition, all these sanitizer techniques are orthogonal to our approach and can be used to improve our fuzzer.

Directed Fuzzing. Different from the randomly fuzzing, directed fuzzing has pre-provided goals such as reaching specific vulnerability locations or digging specific type vulnerability. Typically, AFLGo[21] and Hawkeye[31] employ distance metrics to direct fuzzing to reach pre-provided vulnerability locations as quickly as possible, thus reproducing vulnerabilities at the specific locations. Different from AFLGo, RegionFuzz is a heuristic-based randomly fuzzing tool and aims to sufficiently explore vulnerable paths within a limited budget. It identifies possible vulnerable paths automatically based on code metrics and distributes more energy for them, in order to strengthen fuzzing these regions. SlowFuzz[19] prioritizes seeds that consume more resources (e.g., CPU, memory), and directs fuzzer to detect algorithmic complexity vulnerabilities, while RegionFuzz still focuses on general vulnerabilities that crash programs.

Optimizing Resources Distribution. STADS[26] models fuzzing as discovery of species. Similarly, Böhme Marcel models coverage-based fuzzing as random walking in the Markov chain[13]. Meanwhile, [38] models the problem of identifying the optimal strategy of concolic testing as a model checking problem of Markov decision processes with cost. Existing fuzzing resources distribution strategies focus on dynamic metrics such as path hit frequency, branch hit number and so on. Typically, AFLFast[13] prioritizes seeds that touch less-frequency paths, and tries to balance the energy assignment between cold paths and hot paths. Similarly, FairFuzz[20] spends more energy on low-frequency branches and CollAFL[22] prioritizes seeds that hit more untouched neighbors. [38] proposes a multi-objective based model together with an efficient sorting algorithm for seed prioritization. TortoiseFuzz[30] proposes coverage accounting, a novel approach for input prioritization with metrics evaluating edges in terms of the relevance of memory corruption vulnerabilities. Different from the above work that only considers the dynamic metrics, RegionFuzz distributes the fuzzing resources by also considering static code metrics. RegionFuzz allocates more resources to regions with higher code metrics values, and strengthens fuzzing these regions that are more likely to be vulnerable.

8 Conclusions

We proposed a code metric-aware greybox fuzzing approach. Four kinds of code metrics (i.e., sensitive, complex, deep and rare reachable degree) are designed and utilized to identify regions that are more likely for a vulnerability to reside. Furthermore, these regions are assigned with more fuzzing energy based on the value of code metrics; thus they could be strengthened fuzzing and sufficiently explored. We integrated the code metrics awareness and implemented a new fuzzer named RegionFuzz. Large-scale evaluations have been performed on the typical benchmark including LAVA-M and Google fuzzer-test-suits. The results showed the effectiveness and efficiency of RegionFuzz. Compared with AFL, its maximum improvement on identifying the first crash and the total crashes is 38% and 27.35% respectively. Furthermore, RegionFuzz has helped us find 11 new bugs and identify three new CVEs. In this paper, the proposed four metrics are used independently. It may achieve better performance by combining them in a proper way. We will perform a further study about the way and performance of combining different metrics in the future work.

References

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 1990, 33(12): 32-44. DOI: 10.1145/96267.96279.

[2] Li J, Zhao B, Zhang C. Fuzzing: A survey. *Cybersecurity*, 2018, 1(1): Article No. 6. DOI: 10.1186/s42400-018-0002-y.

[3] Sutton M, Greene A, Amini P. Fuzzing: Brute Force Vulnerability Discovery (1st edition). Addison-Wesley Professional, 2007.

[4] Chen C, Cui B, Ma J, Wu R, Guo J, Liu W. A systematic review of fuzzing techniques. *Computers & Security*, 2018, 75: 118-137. DOI: 10.1016/j.cose.2018.02.002.

[5] Manès V J M, Han H S, Han C, Cha S K, Egele M, Schwartz E J, Woo M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Software Engineering*. DOI: 10.1109/TSE.2019.2946563.

[6] Devarajan G. Unraveling SCADA protocols: Using sulley fuzzer. In *Proc. the DEF CON 15 Hacking Conf.*, August 2007.

[7] Gascon H, Wressnegger C, Yamaguchi F, Arp D, Rieck K. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Proc. the 11th International Conference on Security and Privacy in Communication Networks*, October 2015, pp.330-347. DOI: 10.1007/978-3-319-28865-9_18.

[8] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. In *Proc. the 31st Int. Software Engineering*, May 2009, pp.474-484. DOI: 10.1109/ICSE.2009.5070546.

[9] Wang T, Wei T, Gu G, Zou W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proc. the 2010 IEEE Symposium on Security and Privacy*, May 2010, pp.497-512. DOI: 10.1109/SP.2010.37.

[10] Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vingna G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proc. the 23rd Annual Network and Distributed System Security Symposium*, February 2016. DOI: 10.14722/ndss.2016.23368.

[11] Godefroid P, Levin M Y, Molnar D. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 2012, 55(3): 40-44. DOI: 10.1145/2093548.2093564.

[12] Situ L, Wang L, Li X, Guan L, Zhang W, Liu P. Energy distribution matters in greybox fuzzing. In *Proc. the 41st Int. Software Engineering: Companion Proceedings*, May 2019, pp.270-271. DOI: 10.1109/ICSE-Companion.2019.00109.

[13] Böhme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. *IEEE Trans. Software Engineering*, 2017, 45(5): 489-506. DOI: 10.1109/TSE.2017.2785841.

[14] Pham V T, Böhme M, Santosa A E, Caciulescu A R, Roychoudhury A. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2019.2941681.

[15] Du X, Chen B, Li Y, Guo J, Zhou Y, Liu Y, Jiang Y. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proc. the 41st Int. Software Engineering*, May 2019, pp.60-71. DOI: 10.1109/ICSE.2019.00024.

[16] Li Y, Su Z, Wang L, Li L. Steering symbolic execution to less traveled paths. *ACM SIGPLAN Notices*, 2013, 48(10): 19-32. DOI: 10.1145/2544173.2509553.

[17] Wang X, Sun J, Chen Z, Zhang P, Wang J, Lin Y. Towards optimal concolic testing. In *Proc. the 40th Int. Conf. Software Engineering*, May 2018, pp.291-302. DOI: 10.1145/3180155.3180177.

[18] Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In *Proc. the 36th Int. Conf. Software Engineering*, May 2014, pp.435-445. DOI: 10.1145/2568225.2568271.

[19] Petsios T, Zhao J, Keromytis A D, Jana S. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proc. the 2017 ACM SIGSAC Conference on Computer and Communications Security*, October 2017, pp.2155-2168. DOI: 10.1145/3133956.3134073.

[20] Lemieux C, Sen K. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proc. the 33rd ACM/IEEE Int. Automated Software Engineering*, September 2018, pp.475-485. DOI: 10.1145/3238147.3238176.

[21] Böhme M, Pham V T, Nguyen M D, Roychoudhury A. Directed greybox fuzzing. In *Proc. the 2017 ACM SIGSAC Conference on Computer and Communications Security*, October 2017, pp.2329-2344. DOI: 10.1145/3133956.3134020.

[22] Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z. CollAFL: Path sensitive fuzzing. In *Proc. the 2018 IEEE Symposium on Security and Privacy*, May 2018, pp.679-696. DOI: 10.1109/SP.2018.00040.

[23] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In *Proc. the 2018 IEEE Symposium on Security and Privacy*, May 2018, pp.711-725. DOI: 10.1109/SP.2018.00046.

[24] Dolan-Gavitt B, Hulin P, Kirda E, Lee T, Mambretti A, Robertson W, Ulrich F, Whelan R. LAVA: Large-scale automated vulnerability addition. In *Proc. the 2016 IEEE Symposium on Security and Privacy*, May 2016, pp.110-121. DOI: 10.1109/SP.2016.15.

[25] Woo M, Cha S K, Gottlieb S, Brumley D. Scheduling blackbox mutational fuzzing. In *Proc. the 2013 ACM SIGSAC Conference on Computer & Communications Security*, November 2013, pp.511-522. DOI: 10.1145/2508859.2516736.

[26] Böhme M. STADS: Software testing as species discovery. *ACM Transactions on Software Engineering and Methodology*, 2018, 27(2): Article No. 7. DOI: 10.1145/3210309.

[27] Situ L Y, Wang L Z, Liu Y, Mao B, Li X. Automatic detection and repair recommendation for missing checks. *Journal of Computer Science and Technology*, 2019, 34(5): 972-992. DOI: 10.1007/s11390-019-1955-3.

[28] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. VUzzer: Application-aware evolutionary fuzzing. In *Proc. the 24th Annual Network and Distributed System Security Symposium*, February 26–March 1, 2017. DOI: 10.14722/ndss.2017.23404.

[29] Klees G, Ruef A, Cooper B, Wei S, Hichk M. Evaluating fuzz testing. In *Proc. the 2018 ACM SIGSAC Conference on Computer and Communications Security*, October 2018, pp.2123-2138. DOI: 10.1145/3243734.3243804.

[30] Wang Y, Jia X, Liu Y, Zeng K, Bao T, Wu D, Su P. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *Proc. the 27th Annual Network and Distributed System Security Symposium*, February 2020. DOI: 10.14722/ndss.2020.24422.

[31] Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, Liu Y. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proc. the 2018 ACM SIGSAC Conference on Computer and Communications Security*, October 2018, pp.2095-2108. DOI: 10.1145/3243734.3243849.

[32] Vargha A, Delaney H D. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 2000, 25(2): 101-132. DOI: 10.3102/10769986025002101.

[33] Arcuri A, Briand L. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 2014, 24(3): 219-250. DOI: 10.1002/stvr.1486.

[34] Li Y, Chen B, Chandramohan M, Lin S W, Liu Y, Tiu A. Steelix: Program-state based binary fuzzing. In *Proc. the 11th Joint Meeting on Foundations of Software Engineering*, August 2017, pp.627-637. DOI: 10.1145/3106237.3106295.

[35] Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: A fast address sanity checker. In *Proc. the 2012 USENIX Annual Technical Conference*, June 2012, pp.309-318.

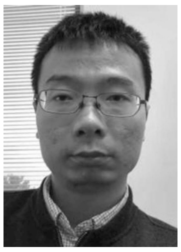
[36] Stepanov E, Serebryany K. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *Proc. the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, February 2015, pp.46-55. DOI: 10.1109/CGO.2015.7054186.

[37] Serebryany K, Iskhodzhanov T. ThreadSanitizer: Data race detection in practice. In *Proc. the Workshop on Binary Instrumentation and Applications*, December 2009, pp.62-71. DOI: 10.1145/1791194.1791203.

[38] Li Y, Xue Y, Chen H, Wu, X, Zhang C, Xie X, Wang H, Liu Y. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In *Proc. the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, August 2019, pp.533-544. DOI: 10.1145/3338906.3338975.

Ling-Yun Situ is an assistant professor in the School of Information Management, Nanjing University, Nanjing. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 2020. His research interests include software and system security, static analysis, fuzzing and deep learning.

Zhi-Qiang Zuo is an associate researcher in Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He got his Ph.D. degree in computer science from National University of Singapore, Singapore, in 2015. His research interests include system software, programming languages, and software engineering.


Le Guan is an assistant professor in the Department of Computer Science at the University of Georgia, Athens. He received his Ph.D. degree in computer science from Chinese Academy of Sciences, Beijing, in 2015. His research interests include mobile security and IoT systems security.

Lin-Zhang Wang is a professor in State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 2005. His research interests include software security testing, model based testing and verification.

Xuan-Dong Li is a professor in State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 1994. His research interests include software modelling, testing and verification.

Jin Shi is a professor in the School of Information Management, Nanjing University, Nanjing. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 2008. His research interests include security and competitive intelligence, academic cooperation behaviour analysis and blockchain security.

Peng Liu is a professor in the College of Information Sciences and Technology at Pennsylvania State University, State College. He received his Ph.D. degree in information technology from George Mason University, VA, in 1999. His research interests include software and system security, IoT security, and AI for security.