2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Critical Variable State-Aware Directed Greybox Fuzzing

1st Xu Chen
IIE,CAS
School of Cyber Security, UCAS
Beijing, China
chenxu@iie.ac.cn

2nd Ningning Cui
IIE,CAS
School of Cyber Security, UCAS
Beijing, China
cuiningning@iie.ac.cn

3rd Zhe Pan
IIE,CAS
School of Cyber Security, UCAS
Beijing, China
panzhe@iie.ac.cn

4th Liwei Chen*
IIE,CAS
School of Cyber Security, UCAS
Beijing, China
chenliwei@iie.ac.cn

5th Gang Shi
IIE,CAS
School of Cyber Security, UCAS
Beijing, China
shigang@iie.ac.cn

6th Dan Meng
IIE,CAS
School of Cyber Security, UCAS
Beijing, China
mengdan@iie.ac.cn

***Abstract*—Directed fuzzing is an effective software testing method that guides the fuzzing campaign towards user-defined target sites of interest, enabling the discovery of vulnerabilities relevant to those sites. However, even though the generated test cases cover the code near the target sites, complex vulnerabilities remain untriggered. By focusing only on test cases that cover new edges, the program states related to the targets are overlooked, resulting in insufficient testing of the targets and failure to capture complex vulnerabilities.**

In this paper, we propose a novel directed fuzzing solution named CSFuzz, which considers program states associated with the targets. First, CSFuzz extracts critical variables related to the target sites from the program using static analysis. Then, CSFuzz monitors the runtime values of these critical variables and infers the program states associated with the targets by adaptively partitioning the range of variable values. This allows CSFuzz to store interesting seeds in the state corpus that trigger new states near the target sites. Lastly, CSFuzz employs dynamic scheduling techniques to guide the fuzzing campaign in selecting different corpora and prioritizing seeds. This ensures more adequate testing of the target sites. We have implemented a prototype of CSFuzz and evaluated it on 2 benchmarks and widely fuzzed real-world software. Evaluation results show that CSFuzz outperforms state-of-the-art fuzzers in terms of vulnerability detection capability, achieving a maximum speedup of 219%. Moreover, CSFuzz has discovered 4 new bugs, including 2 CVE IDs assigned.

***Index Terms*—Fuzzing, Directed Testing, Software Testing**

I. Introduction

Fuzzing is an effective automated program testing technique for discovering vulnerabilities, and it has had a significant impact on both the industry [1]–[4] and the academic community [5]–[8]. By mutating a selected seed to generate numerous test cases for executing the program under test (PUT), fuzzing can easily and effectively discover program bugs when crashes occur [9]. Fuzzing techniques can be classified as blackbox [10], whitebox [11]–[13], and greybox [4], [14] based on the amount of information obtained from program internals. Greybox fuzzing leverages lightweight instrumentation to obtain partial runtime program information, striking a balance between efficiency and overhead. Therefore, greybox fuzzing has gained wide research and adoption [1], [4], [6], [14].

Although greybox fuzzing has been widely researched and utilized, it treats all code regions equally without specifically targeting important sites of interest. To deal with this issue, Böhme et al. proposed AFLGo [15] in 2017, which is the pioneer of Directed Greybox Fuzzing (DGF). The goal of DGF is to spend more time on reaching and testing target sites, rather than wasting resources on unrelated code areas. To guide the fuzzing campaign towards target sites, DGF requires determining the proximity between seed execution and target sites. During the fuzzing process, energy is allocated to seeds based on their scores. Therefore, instead of maximizing code coverage like traditional greybox fuzzing, DGF can reach and test the target sites in the PUT more quickly. Because of these features, DGF is more suitable and performs better in patch testing [16], verifying static analysis reports [17], and crash reproduction [18].

In order to reach target sites more effectively and trigger corresponding vulnerabilities, DGF approaches have made efforts. Hawkeye [19] addresses the distance bias issue of AFLGo by introducing diverse distance strategies to evaluate the execution traces of seeds. FishFuzz [20] utilizes the distance from the target functions as vector-based indicators. WindRanger [21] introduces the concept of deviation of basic blocks and utilizes them to measure distances. Titan [22] optimizes input generation by considering the correlation between targets in the program. AFLRun [23] maintains an extra virgin map for each target to incorporate path diversity and unbiased energy allocation for the targets.

However, the current DGF falls short of expectations in terms of its effectiveness in discovering vulnerabilities. This is because, under normal circumstances, executing the PUT

[1]Corresponding author

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00219

requires reaching specific program states to penetrate the code and reach the target sites. Alternatively, even if the target code is executed, it may not trigger the corresponding vulnerability because specific program states are required to satisfy the triggering conditions. For example, suppose a test case S is executed by the PUT at runtime and will be discarded if it does not trigger new edge coverage. But if S causes a new program state, which is helpful for detecting target vulnerabilities, it will be an interesting test case. To enhance the effectiveness of DGF in discovering these vulnerabilities, we need to focus on these critical program states.

First of all, **what is a program state?** Essentially, programs store data in variables, and at any given point of program execution, the stored contents are referred to as the program's state. However, the large number of variables in a program poses an unacceptable performance overhead when considering all variables, impeding the progress of the fuzzing effort. As a result, striking a balance between effectiveness and performance overhead requires focusing on a subset of states. Additionally, **which program states should DGF track?** To improve the effectiveness of triggering potential vulnerabilities, we need to pay more attention to states that are relevant to the target sites. Firstly, we consider the distance of variables to the target sites and their reachability. In static single assignment languages, each variable can only be assigned once. This implies that before triggering a vulnerability, some variables that influence the target sites but are distant will be assigned to new variables before use. Consequently, variables that are distant from the crash sites have minimal or no impact on the desired program states. Simultaneously, we exclude variables that lack a path to reach the target sites, as these variables cannot impact the target sites. Secondly, we consider the location of the target site's related memory. The value of a variable that records a particular state is often kept in memory. Some of these values directly or indirectly influence branches. Furthermore, bugs that violate memory safety in software written in memory unsafe languages frequently have security implications [24], [25]. AddressSanitizer (ASan) is a memory error detector that can record invalid memory read and write operations [26]. Therefore, we consider the values of variables related to ASan detection to be more closely associated with target-related states, and considering these relevant states is beneficial for triggering the corresponding vulnerabilities.

To identify program states related to the targets and guide DGF, we propose CSFuzz, a directed greybox fuzzing approach aware of critical variable states. First, through static analysis, we extract relevant variables based on their distance to the target sites and reachability. Then we leverage ASan annotations as indicators to identify critical variables. At runtime, we monitor the values of these critical variables and adaptively divide their values into intervals based on additional recorded $init\_value$. This enables us to capture program state behaviors with additional feedback as a supplement to code coverage. We use an additional state corpus to store test cases that discover new states, enabling a more thorough exploration of the program's areas of interest. We also use a dynamic seed schedule strategy to enhance the performance.

```
static int fill_buffer_resample(lame_internal_flags * gfc,
     sample_t * outbuf,...)
{
    int   BLACKSIZE, xvalue;
    int  *inbuf_old;
    ......
    BLACKSIZE = filter_l + 1;
    if (gfc->buffer_resample == 0)
    {
          ......
          y = inbuf_old[BLACKSIZE + j2];  //bug location
          xvalue += y
          ......
          outbuf[k] = xvalue;
    }
    else
    {
      ......
    }
}
```

Listing 1. A simplified motivation example

We implemented CSFuzz on the general-purpose fuzzing framework AFL [4] to demonstrate its effectiveness and scalability. To evaluate the performance of CSFuzz, we selected Unibench [27] and Magma [28] as benchmarks and compared it with state-of-the-art fuzzers including AFL [4], AFL++ [1], FishFuzz [20], AFLGo [15], WindRanger [21], and the CmpLog mode of AFL++. We conducted 10 rounds of testing for each program and calculated the average values to assess crash reproduction and vulnerability discovery capabilities. The experimental results indicate that compared with AFL, AFL++, FishFuzz, AFLGo, WindRanger, and the CmpLog mode of AFL++, CSFuzz achieved an average speedup of 152%, 107%, 119%, 86%, 88% and 81% in triggering target crashes. Moreover, CSFuzz uncovered previously unknown vulnerabilities in widely tested open-source software such as gpac [29] and libxml2 [30], leading to the allocation of two CVE IDs.

We made the following contributions:

- We propose the concept of using critical variables to track program states and a static analysis method to identify critical variables.
- We propose a state feedback mechanism to help DGF efficiently explore targets.
- We design a seed scheduling strategy to improve the efficiency of DGF.
- We implement a prototype of CSFuzz, comparing it with the state-of-the-art fuzzing methods, and confirming its superiority on real-world benchmark programs.

II. Motivation Example

In this section, we use a code snippet from `lame` as our motivation example to illustrate and discuss the limitations of other directed fuzzing methods for finding bugs in such scenarios (§ II-A). Additionally, we investigate how CSFuzz aids in successfully detecting them (§ II-B).

A. Fuzzing Scenario

Listing 1 shows a snippet of code that deals with buffer data and contains a potential bug. It processes elements in `inbuf_old` and `outbuf`. The condition to reach the bug area is met only if `buffer_resample` is equal to zero.

Assuming before the directed fuzzing campaign, the target sites that include the bug are set. The fuzzing process uses code coverage to keep interesting seeds in the corpus. To improve directionality, DGF prioritizes seeds by considering those that are closer to the target sites. However, in some cases, test cases that have new program states but do not provide new code coverage are discarded by the fuzzer, even if they would be more helpful in advancing the fuzzing process.

Let's consider a scenario where a DGF generates a test case A that reaches line 7 with new edge coverage for the first time. Although it doesn't meet the conditions to go further into the code, test case A is kept in the corpus due to code coverage feedback [4]. As the fuzzing continues, suppose a new test case B reaches line 7 with a different `buffer_resample` value. Test case B creates a new program state but does not provide any new edge coverage feedback, so it is discarded by the fuzzer. However, this state reveals new behavior in the variables affecting the target site, which is actually very interesting.

Similarly, when the fuzzing generates a test case C that first reaches the target site, this new seed with edge coverage feedback is retained. However, since there is no out-of-bounds access at line 10, an error may not necessarily occur. If a new test case D produces a different value at line 10, especially with changes in `BLACKSIZE + j2`, it creates a program state closely related to triggering bugs. However, without new edge coverage feedback, this seed is discarded instead of being kept in the corpus. This results in missed opportunities for further fuzzing and insufficient testing in the target area.

B. Solutions

It is essential to capture important program behaviors and test target sites more comprehensively. The challenge lies in identifying and focusing only on the important variables in the program while also avoiding excessive resource consumption in unrelated areas and state explosions.

In CSFuzz, we filter variables based on their distance to the target sites and reachability, selecting those that are more relevant to the target sites. Additionally, we focus on variables that are more closely associated with triggering vulnerabilities based on ASan sanitizer indications. Based on this, we identify a set of variables that help trigger vulnerabilities related to the targets, such as the variables `buffer_resample` and `BLACKSIZE + j2` in the program.

To achieve more comprehensive program testing, we monitor the runtime values of critical variables during the execution of the PUT. By adaptively partitioning the range of these variables, we can preserve seeds that trigger new program states. Even if no new edge is triggered but a new program state emerges, we can retain the corresponding test case in the corpus for further testing at a later time. Therefore, when there are changes in the values of `buffer_resample` or `BLACKSIZE + j2`, these values will be captured and considered as triggering different program behaviors. The corresponding test cases will be preserved for further testing, thereby increasing the likelihood of triggering vulnerabilities related to this target site.

III. CSFuzz Approach

Fig. 1 illustrates the complete workflow of CSFuzz, which consists of two main parts: static analysis and fuzzing. In the static analysis phase of CSFuzz, the source code is compiled into the LLVM IR intermediate file. The critical variables within the program are subsequently identified and screened by synergistically combining the provided target sites of interest. Simultaneously, distance information is calculated similarly to FishFuzz [20], prioritizing a set of seeds that are closer to the targets to guide the fuzzing campaign towards the target sites. In addition to collecting edge coverage through instrumentation like other fuzzers, CSFuzz also incorporates instrumentation to track critical variables to monitor the runtime state of the PUT.

As the fuzzing loop continues, CSFuzz dynamically partitions the range of critical variables based on feedback information. When a test case triggers new edge coverage, it is added to the edge coverage corpus. In cases where there is no new edge coverage but the variable's value falls within the previously uncovered range, it is added to the state corpus. CSFuzz then proceeds to select seeds for the next cycle. It applies a strategy to choose the corpus from which seeds will be selected for the subsequent stage. Regarding the edge corpus, CSFuzz works like a normal DGF and appropriately switches between the exploration stage and the exploitation stage. For the state corpus, CSFuzz prioritizes seeds and determines their selection order.

In the following sections, we will explain following issues in detail: how does CSFuzz extract critical variables using static analysis (§ III-A); how does CSFuzz divide the range of variable values to monitor the running state of the program (§ III-B); how does CSFuzz apply corpus selection strategy (§ III-C); and how does CSFuzz schedule seeds to determine queue priority (§ III-D).

A. Critical Variable Recognition

Identifying critical variables in a program is a primary objective of CSFuzz during static analysis. Extracting an excessive number of variables can result in adding an excessive number of seeds to the state corpus at runtime, thereby reducing the efficiency of fuzzing. Conversely, if critical variables are overlooked during extraction, it may lead to insufficient exploration of important program states. Striking the right balance between these two aspects is crucial. To address this challenge, we propose two strategies. Strategy 1 involves identifying variables that are closely related to the target sites and strategy 2 focuses on variables associated with ASan checks. Variables that meet both strategies' criteria are considered candidate critical variables. Subsequently, we filter

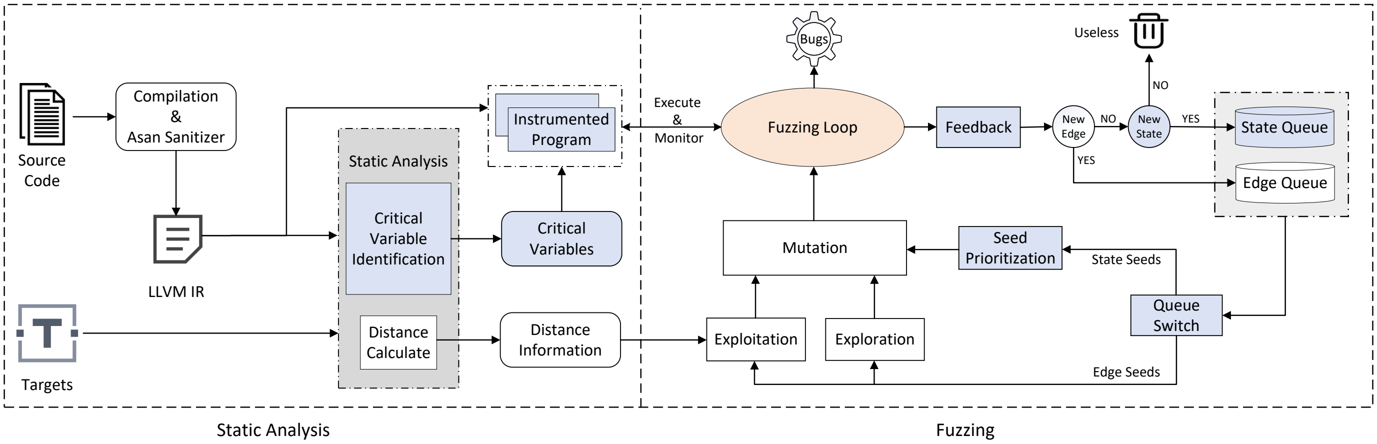

Fig. 1. Overview of CSFuzz workflow

the candidate variables and obtain the critical variables specific to the PUT. Our analysis is conducted at the LLVM IR level, which is a static single assignment language and each variable can only be assigned once in the program.

Strategy 1. For a given program p and target sites ts, the variables in program p that satisfy strategy 1 are defined as follows:

$$S_1(p) = \{v \in ALLV(p) \mid (\exists t \in ts)[calldist(F(v), F(t)) < h \wedge isreachable(v,t)]\} \;, \tag{1}$$

where $S_1(p)$ represents the variables in program p that satisfy strategy 1; $ALLV(p)$ refers to all variables in p; $F(v)$ indicates the function in which variable v is located; $F(t)$ indicates the function in which the target t is located; $calldist()$ calculates the shortest call distance between two functions; h is a threshold; $isreachable(v,t)$ indicates whether variable v can reach target t through the shortest function calls. For example, if instruction I in function A calls the function B, the shortest call distance between the two functions is 1. If the function B contains an instruction that calls the function C, and there is no instruction in function A that calls C, then the shortest call distance between A and C is 2. If v is in function A and t is in function B, we consider $isreachable(v,t)$ to be true when the basic block containing v has a path to the basic block containing the calling instruction I.

In strategy 1, if there is a target t in ts, such that the call distance between the function where variable v resides and the function where the target t resides is within a threshold value, and v can reach the call site within the function, then v is considered to satisfy strategy 1. In other words, the goal of strategy 1 can be explained as selecting variables that are closer to the target sites. This is because these variables have a stronger correlation with the targets, making a greater contribution to triggering related vulnerabilities. Furthermore, CSFuzz utilizes SVF analysis [31] to handle indirect calls and can also select other functions as supplements to strategy 1, enhancing robustness and scalability.

```
cond.true:                                    ; preds = %for.body127
  %316 = load i32*, i32** %inbuf_old
  %317 = load i32, i32* %BLACKSIZE
  %318 = load i32, i32* %j2
  %add133 = add nsw i32 %317, %318
  %idxprom134 = sext i32 %add133 to i64
  %arrayidx135 = getelementptr inbounds i32, i32* %316, i64 %idxprom134
  %319 = ptrtoint i32* %arrayidx135 to i64
  %320 = lshr i64 %319, 3
  %321 = add i64 %320, 2147450880
  %322 = inttoptr i64 %321 to i8*
  %323 = load i8, i8* %322
  %324 = icmp ne i8 %323, 0
  br i1 %324, label %325, label %331

325:                                    ; preds = %cond.true
  %326 = and i64 %319, 7
  %327 = add i64 %326, 3
  %328 = trunc i64 %327 to i8
  %329 = icmp sge i8 %328, %323
  br i1 %329, label %330, label %331

330:                                    ; preds = %325
  call void @__asan_report_load4(i64 %319) #13
  unreachable

331:                                    ; preds = %325, %cond.true
  %332 = load i32, i32* %arrayidx135
  br label %cond.end
```

Fig. 2. LLVM IR with ASan instrumentation

Strategy 2. Building upon the variables that satisfy strategy 1, the variables satisfying strategy 2 are defined as follows:

$$S_2(v) = \{v \in S_1(p) \mid rel(v, asan) \vee rel(getop(v), asan)\} \;, \tag{2}$$

where $S_2(v)$ represents the variables that satisfy strategy 2; $asan$ refers to the sanitizer identifier used for runtime error detection in the program; rel corresponds to the relevant checks; and $getop(v)$ retrieves the operand of the instruction corresponding to v.

In other words, the goal of strategy 2 can be explained as selecting variables that are associated with ASan checks. Variables that satisfy strategy 2 are more likely to involve states related to the targets and trigger corresponding vulnerabilities.

As shown in Fig. 2, assuming these variables satisfy strategy 1, the code enclosed in dashed boxes represents the vulnerability check code instrumented by the sanitizer. The variable `%319` is monitored by the ASan sanitizer, and `%319` is derived from `%arrayidx135` in the program. Therefore, we consider that the `%arrayidx135` satisfies strategy 2. Similarly, `%332` has `%arrayidx135` as its operand, and since `%arrayidx135` is monitored by ASan, we also consider `%332` satisfies strategy 2.

Algorithm 1 Critical variable identification

Input: A set of targets: Ts
Input: Instrumented program: P
Output: Critical variables: CVs
1: $fs \leftarrow$ GetFunc(Ts)
2: **for** f in P **do**
3:   **for** ft in fs **do**
4:     $dis \leftarrow$ Calldist(f, ft)
5:     **if** $dis < h$ **then**
6:       $cs \leftarrow$ CallSite(f, ft)
7:       **for** v in f **do**
8:         **if** reachable(v, cs) **then**
9:           $v1 \leftarrow v1 \cup v$
10:         **end if**
11:       **end for**
12:     **end if**
13:   **end for**
14: **end for**
15: **for** v in $v1$ **do**
16:   **if** rel($v, asan$) || rel($getop(v), asan$) **then**
17:     $CVs \leftarrow CVs \cup v$
18:   **end if**
19: **end for**
20: $CVs \leftarrow$ PointerOrInteger(CVs)

Based on strategy 1 and strategy 2, we have identified candidate critical variables that have stronger correlations with the target sites and a higher potential for triggering vulnerabilities. To further improve the efficiency, we select pointer type and integer type variables from these candidates. This selection is based on observations that program states are often represented by integer values, and vulnerabilities are often related to pointers.

Algorithm 1 shows the process of selecting critical variables from the program. After obtaining the functions fs where the targets are located (Line 1), we can traverse the function f in P and ft in fs to calculate the call distance from f to ft (Line 2-4). If the distance from f to the function ft is less than the threshold h(default is 3), we identify the call site cs in f (Line 5-6). If the variable v in f has a path to reach this call site, we consider it satisfying strategy 1 (Lines 7-9). Additionally, if the variable v is also related to ASan sanitizer, we consider it to satisfy both strategy 1 and strategy 2 (Lines 15-17). Finally, by filtering for pointer type and integer type variables, we obtain the critical variables in the program (Line 20).

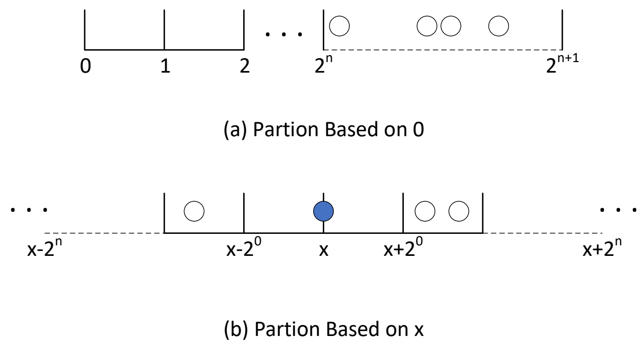

Fig. 3. Variable range partition

B. State Feedback

To capture the runtime program state, we perform additional instrumentation on the critical variables in the program. In AFL, the basic blocks are instrumented and shared memory is used to record the edge coverage of the currently executing seed. It also uses a virgin map to store the coverage information of all previously executed inputs. Similar to AFL, we assign a unique ID to each critical variable, which allows us to record the variable's value at runtime in the corresponding location in shared memory. To achieve uniformity and balance performance overhead, we standardize the feedback data to 32 bits through instrumentation. For data below 32 bits, we perform zero extension to bring it to 32 bits. For data above 32 bits, we truncate the higher-order bits and retain the lower 32 bits. This is coarse-grained but useful for fuzzing.

Subsequently, the fuzzer reads data from shared memory to obtain the runtime information of the PUT. Utilizing this runtime information, it is necessary to determine if the current test case triggers a new interesting program state. The value of each critical variable has the potential to hold important program state information, but it is impractical to keep track of all states. Therefore, to avoid state explosions, we attempt to partition the variables into ranges. If the value of a variable falls within a range that has not been previously recorded, it is considered to trigger a new state.

A simple partition approach is to adopt AFL's power-of-two storage mechanism for numerical values, such as [1, 2), [2, 4), [4, 8), and subsequent intervals. However, this straightforward partitioning may lead to inaccurate monitoring. As shown in Fig. 3 (a), different variables have different range distributions, and when a variable is concentrated within one large range, detecting the state of the variable becomes ineffective. To address this issue, we have implemented an adaptive method that assigns different range intervals to each variable, enabling more accurate state detection. At runtime, an initial value is recorded for each variable. When a variable's value appears for the first time, the initial value is initialized with the corresponding value. As shown in Fig. 3 (b), subsequent range partitioning of variables is based on the initial value

x, allowing for adaptive range interval assignment to different variables.

Although some variables have unpredictable distributions, exploring different behaviors of variables at runtime complements edge coverage. For variables with distribution patterns that are often used to represent specific states, we utilize initial values to determine their range distribution. In CSFuzz, this approach allows for more effective capture of behavioral changes. If the execution of a test case does not trigger new edge coverage, but the tracing covers any new bit in the state virgin map that is not covered by the previous corpus, it indicates that the variable's value falls within a new range. In such cases, we consider it to trigger a new program state. Then the state virgin map is updated, and the test case is stored in the state corpus. This method of adaptively partitioning the range of variables to determine states not only avoids state explosion but also ensures more precise state detection for critical variables.

C. Corpus Selection

To better maintain seeds that trigger new edges or states, we utilize two corpora to store interesting seeds. When edge coverage is triggered, the seed is saved in corpus C_1. If there is no new edge coverage, but a new program state is triggered, the seed is saved in corpus C_2. By balancing coverage exploration and state exploration, we assist DGF in reaching the targets and triggering the corresponding vulnerabilities as quickly as possible. To effectively choose a seed for the next iteration of fuzzing, we employ a probability p to determine which corpus C to select from.

$$C = \begin{cases} C_1, & \text{if } r < p \\ C_2, & \text{else .} \end{cases} \tag{3}$$

If the random value r between 0 and 1 is less than p, we choose the next loop using a seed from corpus C_1. Otherwise, we choose a seed from corpus C_2. The probability p is dynamically updated at runtime at regular intervals by δ, based on the reward values for selecting different corpora.

$$p = \varphi(p + \delta) \ , \tag{4}$$

$$\delta = \left(\frac{a \cdot m_1}{NC_1} - \frac{b \cdot m_2}{NC_2} \right) \cdot (NC_1 + NC_2) \ , \tag{5}$$

where a and b represent two parameters; m_1 represents the average reward value of selecting the C_1 corpus over the past rounds. The reward value stands for the number of test cases added to the C_1 corpus by mutating seeds from C_1. Similar to m_1, m_2 corresponds to the C_2 corpus; NC_1 and NC_2 correspond to the number of seeds in the two corpora. A corpus with fewer seed numbers implies that the addition of corresponding seeds is relatively rarer, so we make the reward value more sensitive when new seeds are added. Additionally, φ is used to adjust the probability p to keep it between 0.1 and 0.9. This ensures the correctness of the probabilities while also avoiding extreme situations. Even when biased towards one corpus, there remains a probability of selecting another corpus to obtain a reward value, gradually adjusting the probability p.

In practical terms, we tend to adjust the probability p based on the rewards of the corpora. Suppose we select seeds from C_1 that result in interesting test cases being added to C_1. Intuitively, we should increase the reward value of C_1 because there are new code regions to be tested. When calculating the reward, we only consider the quantity of seeds added to the corresponding corpus. This is because if selecting C_1 generates interesting test cases added to C_2, these test cases are responsible for exploring program states rather than exploring new code regions. Therefore, they should not contribute to the reward of C_1.

CSFuzz selects a corpus based on dynamic probability. This involves exploring new code regions and assisting in guided fuzzing towards target sites during the directed stage, and also explores program states and conducts more comprehensive testing of the targets.

D. Seed Prioritization

The chosen code coverage corpus works like other DGF methods, which are divided into exploration and exploitation stages. We used a fixed probability to switch the stage between exploration and exploitation for the code coverage corpus. In the exploration stage, CSFuzz focuses on enough coverage exploration to mitigate the risk of getting stuck in local optima. In the exploitation stage, for each target, CSFuzz assigns a higher selection probability to a set of seeds that are closer to the target. In the next round of seed selection, the higher the priority, the higher the probability of being selected.

When choosing the state corpus, the seeds in the state corpus trigger new states relevant to the target sites, representing the execution of seeds around the target sites. Therefore, we primarily consider two main aspects to prioritize the seeds within the corpus. (1) We tend to prioritize selecting seeds that have lower fuzzing counts among the covered critical variables. In the case of critical variable fuzzing counts, if a seed s covers critical variables a and b, and we select seed s for the fuzzing test, the fuzzing counts of critical variables a and b will increase by one. (2) In the current state corpus, we tend to select the seed that was added to the queue recently. Calculate the probability $P(x)$ of selecting seed x as:

$$p(x) = \frac{1 - a}{min(\xi(x)) \cdot \sum_{i=1}^{n} \frac{1}{n(v_i)}} + \frac{a \cdot t_x}{\sum_{i=1}^{m} t_i} \ , \tag{6}$$

where a is a parameter less than one and greater than zero; $min(\xi(x))$ represents the minimum number of fuzzing counts among the variables covered by seed x; $n(v_i)$ represents the number of fuzzing counts of variable v_i; t_x refers to the time elapsed between the start of the fuzzing campaign and the addition of seed x to the corpus; and t_i summation refers to the sum of the time of addition of all state seeds.

We tend to choose variables with fewer fuzzing counts because it indicates that the variable has not been sufficiently tested. And we also tend to choose the one added to the

TABLE I
UNIBENCH BENCHMARK RESULTS

Project	Bug Id	AFL		AFL++		FishFuzz		AFLGo		WindRanger		CmpLog		CSFuzz	
		TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE
jq	CVE-2015-8863	-	16h52m	-	33h20m	-	12h22m	-	8h19m	-	6h4m	-	31h10m	-	*5h47m
nm-new	CVE-2018-10373	T.O.	T.O.	43h3m	44h19m	40h32m	T.O.	T.O.	T.O.	T.O.	T.O.	23h4m	T.O.	*22h55m	*29h7m
nm-new	CVE-2018-12641	19h45m	T.O.	10h48m	T.O.	7h37m	T.O.	9h5m	T.O.	*3h46m	*26h5m	8h44m	T.O.	5h21m	37h1m
tcpdump	CVE-2017-5203	-	T.O.	-	39h28m	-	35h48m	-	44h40m	-	35h35m	-	34h39m	-	*33h42m
objdump	CVE-2017-9755	1h22m	2h40m	1h48m	8h20m	1h58m	3h19m	2h47m	5h5m	1h6m	*1h49m	1h23m	5h56m	*1h5m	2h43m
objdump	CVE-2017-9755	-	39h40m	-	38h3m	-	38h6m	-	36h13m	-	41h29m	-	*8h16m	-	33h41m
objdump	CVE-2017-7224	-	T.O.	-	T.O.	-	T.O.	-	*46h35m	-	T.O.	-	T.O.	-	T.O.
imginfo	CVE-2017-5500	T.O.	T.O.	T.O.	T.O.	28h27m	28h27m	T.O.	T.O.	T.O.	T.O.	44h35m	44h35m	*12h6m	*12h6m
wav2swf	CVE-2017-1000182	-	T.O.	-	T.O.	-	T.O.	-	T.O.	-	T.O.	-	T.O.	-	*12h42m
wav2swf	CVE-2017-11099	T.O.	T.O.	16h31m	T.O.	12h47m	T.O.	T.O.	T.O.	T.O.	T.O.	22h15m	T.O.	*9h51m	*11h49m
lame	CVE-2015-9101	17m	17m	24m	1h6m	16m	24m	15m	21m	12m	16m	27m	28m	*10m	*14m
lame	CVE-2017-11720	-	1h13m	-	1h16m	-	1h17m	-	1h9m	-	3h43m	-	55m	-	*51m
lame	CVE-2017-15046	11h21m	11h21m	7h38m	9h13m	12h41m	13h49m	7h12m	9h1m	15h	15h	*3h48m	*7h	6h55m	8h53m
sqlite3	CVE-2019-19646	T.O.	T.O.	T.O.	T.O.	*33h46m	*33h46m	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	35h25m	35h25m
sqlite3	CVE-2015-3416	T.O.	T.O.	T.O.	T.O.	*40h2m	*40h2m	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
sqlite3	CVE-2019-19926	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	43h13m	43h13m	T.O.	T.O.	*39h52m	*39h52m
Avg. Speedup		+129%	+85%	+83%	+140%	+50%	+71%	+113%	+90%	+95%	+86%	+66%	+98%	/	/
Avg. Bugs		4.7(+104%)		5.4(+78%)		6.6(+45%)		5.5(+74%)		6.4(+50%)		6.2(+54%)		9.6(/)	

queue recently because it indicates that the seed triggered a new variable, or triggered a rarer variable state. Using the seed selection probability, we choose the next favored seed for mutation.

IV. Implementation And Evaluation

The components of CSFuzz mainly include static analysis and fuzzing. In the static analysis phase, we compile the source code into an LLVM IR file. Based on this, we perform static analysis on the LLVM IR level to extract critical variables. The handling of critical variables is implemented by an LLVM pass, which consists of 1300 lines of C++ code. As for the fuzzing phase, the prototype of CSFuzz is based on AFL version 2.57b, with 1200 lines of code implementing component functionality.

To demonstrate the effectiveness of CSFuzz, we conducted experiments to answer the following questions:

- **RQ1.** How effective is CSFuzz in reproducing vulnerabilities compared to other fuzzing techniques?
- **RQ2.** What is the impact of each component of CSFuzz on the overall effectiveness?
- **RQ3.** Will the state feedback of CSFuzz bring too much overhead or state explosion?
- **RQ4.** Can CSFuzz help in triggering real-world vulnerabilities?

A. Evaluation Setup

Evaluated Techniques. We compared CSFuzz with 6 state-of-the-art fuzzing tools. CSFuzz is the prototype of the method proposed in this paper. We set various hyperparameters based on experience and kept them fixed during the experiments. AFL [4] is a classic greybox fuzzer with edge coverage guidance, and many fuzzers [6], [15], [32], [33] are implemented based on the AFL framework. AFL++ [1] is an open-source project that is constantly being updated, integrating the better features developed over the years for the AFL series of fuzzers. FishFuzz [20] is one of the most advanced directed greybox fuzzers, guiding the fuzzing towards target sites based on call distance between functions and maximizing code coverage. We chose the FishFuzz prototype based on the AFL implementation. AFLGo [15] is a kind of directed greybox fuzzing, which has pioneered directed fuzzing. Some works are based on AFLGO [21], [23], [34]. WindRanger [21] employs deviation basic blocks to facilitate DGF, representing one of the most advanced DGF technologies. The CmpLog mode of AFL++ [1] is inspired by input-to-state correspondence in REDQUEEN [35]. It helps address path constraints but is not specific to any particular target site. Other directed greybox fuzzers (e.g. HawkEye [19], Titan [22], AFLRun [23], etc.) are not open-source at the time of writing this paper.

Datasets. We utilize the Unibench dataset [27] and the Magma dataset [28]. The Unibench dataset provides real-world programs for evaluating fuzzers, which can be divided into 6 categories according to the input format of the PUT. We selected 8 programs covering all categories for testing. The Magma dataset consists of nine projects that manually inserted previously existing CVE vulnerabilities into a patched program to evaluate the performance of fuzzers. We excluded two projects that failed to compile and deployed the rest according to the benchmark instructions.

Configuration. To evaluate the performance of a directed fuzzer in crash reproduction testing, it is necessary to set target sites. We obtained target sites for the Unibench projects by running some PoC inputs stored in the MITRE CVE database [36], and for Magma by using PoC inputs stored in Magma

TABLE II
MAGMA BENCHMARK RESULTS

Bug Id	AFL		AFL++		FishFuzz		AFLGo		WindRanger		CmpLog		CSFuzz	
	TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE	TTR	TTE
XML009	-	T.O.	-	37h	-	7h41m	-	16h25m	-	10h52m	-	35h50m	-	*6h2m
XML017	-	*11m	-	50m	-	1h14m	-	45m	-	47m	-	15m	-	49m
SSL001	12m	9h15m	5m	6h56m	25m	29h	8m	8h35m	6m	6h33m	8m	8h58m	*4m	*5h22m
SND017	22m	*1h2m	18m	3h40m	27m	20h52m	15m	4h9m	10m	2h44m	14m	1h16m	*5m	1h48m
SND020	22m	T.O.	18m	4h56m	27m	9h12m	15m	5h6m	10m	3h11m	14m	6h38m	*5m	*1h23m
SQL002	13m	9h35m	9m	6h41m	41m	19h23m	9m	8h7m	5m	5h36m	5m	5h35m	6m	*4h19m
SQL014	23h44m	T.O.	14h12m	20h2m	22h53m	T.O.	15h47m	T.O.	*9h16m	*13h17m	12h43m	T.O.	10h24m	14h22m
SQL018	54m	10h38m	1h8m	13h21m	33m	*6h41m	42m	8h59m	*31m	7h51m	1h19m	15h13m	37m	6h54m
LUA003	-	T.O.	-	T.O.	-	*43h34m	-	T.O.	-	T.O.	-	T.O.	-	T.O.
LUA004	30h12m	32h47m	20h25m	23h32m	32h22m	33h31m	33h41m	41h27m	26h29m	32h19m	24h13m	25h26m	*19h45m	*22h20m
TIF002	43h12m	43h12m	*36h16m	*36h16m	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	42h55m	42h57m	38h13m	38h24m
TIF007	2m	4m	2m	4m	3m	4m	7m	16m	12m	34m	2m	*2m	2m	4m
TIF008	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	42h22m	42h22m	*37h43m	*40h16m
TIF012	-	*49m	-	58m	-	1h20m	-	1h26m	-	4h9m	-	2h46m	-	1h28m
TIF014	2m	2h47m	2m	*58m	3m	5h21m	7m	2h14m	12m	4h39m	*1m	2h44m	2m	1h8m
PDF016	4m	12m	*1m	8m	4m	8m	3m	8m	4m	14m	2m	*2m	4m	8m
PDF010	15m	38m	23m	1h28m	7m	44m	11m	31m	9m	24m	9m	26m	*6m	*14m
PDF018	T.O.	T.O.	23h40m	23h40m	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	25h28m	25h28m	*22h46m	*22h46m
PDF021	4m	T.O.	3m	*25h30m	4m	T.O.	4m	T.O.	4m	T.O.	3m	33h12m	4m	T.O.
PDF006	13m	T.O.	12m	T.O.	15m	T.O.	11m	T.O.	12m	T.O.	*6m	T.O.	10m	*46h13m
Avg. Speedup	+97%	+219%	+58%	+73%	+155%	+166%	+87%	+82%	+91%	+90%	+27%	+65%	/	/
Avg. Bugs	11(+40%)		14(+10%)		12(+28%)		13.3(+16%)		13.6(+13%)		14.1(+9%)		15.4(/)	

testing results artifacts [37]. The setting of the initial seed corpus affects the validity. To avoid inconsistency, on Unibench, our initial corpus was selected from the test cases provided by AFL. If there was no corresponding file format, we used a simple file that meets the input of the PUT as the initial corpus. On the Magma dataset, we selected the seeds provided in the Magma benchmark and used them as the initial corpus after removing the inputs that caused the program to crash. We added the ASan sanitizer for runtime detection during program compilation. We also disabled address randomization to allow CSFuzz to better detect the runtime states of pointer variables. Our time limit for each round was 48 hours by default, and we conducted 10 rounds, averaging the results. Our experimental environment was conducted on an Ubuntu 18.04 machine equipped with an Intel Xeon(R) Silver 4216 CPU featuring 64GB of RAM. Under the same configuration, each fuzzer was assigned a core to run in a docker container [38].

B. Bug Reproducing Capability (RQ1)

One of the main application scenarios of directed fuzzing is effectively reproducing vulnerabilities. Therefore, we compare CSFuzz with 6 state-of-the-art fuzzing tools on 2 test datasets using the Time-to-Reach(TTR), Time-to-Exposure (TTE), and the average number of triggered bugs as the evaluation metrics. TTR indicates the duration a fuzzer takes to generate the first input that reaches the target, while TTE indicates the time it takes for a fuzzer to trigger a bug. During the fuzzing test run, if bugs are triggered by all fuzzers within 10 minutes, we exclude the quickly triggered bugs, as they do not prove the validity of the fuzzing test. On the Unibench dataset, when a vulnerability is triggered, we use the logs recorded by the ASan sanitizer, specifically the line of code where the crash occurred and the last three call stacks, to determine whether the target vulnerability has been triggered. The bugs on the Magma dataset are manually inserted, so the trigger of each bug is indicated by a unique ID.

The results of the Unibench dataset are presented in Table I. Symbol (-) indicates that running the initial corpus can reach the corresponding target site. T.O. indicates that the corresponding bug is not reached or triggered within the specified time by the repeatedly executed fuzzer. We calculate averages to obtain TTR and TTE. Specifically, in cases where the corresponding bug is not reached or triggered within the specified time, we use a timeout parameter (48 hours) as a substitute to calculate the average TTR and TTE. The shortest mean TTR and TTE are marked with an asterisk.

The results show that CSFuzz demonstrates significant improvement for most targets. Compared with state-of-the-art fuzzers, CSFuzz performs 89% speed improvement in TTR, 95% speed improvement in TTE, and detects 67% more bugs on average. In general, CSFuzz achieves a mean speedup in TTR of 129% against AFL, 83% against AFL++, 50% against FishFuzz, 113% against AFLGo, 95% against WindRanger, and 66% against CmpLog. It also achieves a mean speedup in TTE of 85% against AFL, 140% against AFL++, 71% against FishFuzz, 90% against AFLGo, 86% against WindRanger, and 98% against CmpLog. The average number of bugs discovered

TABLE III
COMPONENTS INFLUENCE

Bug Id	CSFuzz	CSFuzz-T	CSFuzz-C	CSFuzz-P	CSFuzz-S	CSFuzz-V
CVE-2015-8863	5h47m	8h29m	6h32m	7h24m	10h30m	6h24m
CVE-2018-10373	29h7m	37h22m	36h41m	37h33m	T.O.	T.O.
CVE-2018-12641	37h1m	41h5m	39h5m	40h43m	39h	39h27m
CVE-2017-5203	33h42m	26h41m	35h2m	34h22m	T.O.	38h30m
CVE-2017-9755	2h43m	9h8m	3h35m	4h22m	4h40m	3h51m
CVE-2017-9754	33h54m	39h26m	32h12m	35h15m	38h39m	35h50m
CVE-2017-5500	12h6m	17h56m	14h27m	15h12m	T.O.	17h13m
Avg.	/	+52%	+14%	+23%	+82%	+27%

TABLE IV
RUNTIME OVERHEAD COMPARISON OF CSFUZZ AND AFL.

Project	AFL	CSFuzz	Overhead
jq	72	65	10%
nm-new	159	153	10%
tcpdump	76	74	3%
objdump	155	125	19%
imginfo	356	338	5%
wav	208	192	8%
lame	38	31	18%
sqlite3	191	185	3%
Avg.	157	145	9%

is 104% more than AFL, 78% more than AFL++, 45% more than FishFuzz, 74% more than AFLGo, 50% more than WindRanger, and 54% more than CmpLog.

Similar to Table I, the evaluation results on the Magma dataset for these fuzzers are shown in Table II. CSFuzz performs 85% speed improvement in TTR, 115% speed improvement in TTE, and detects 19% more bugs on average. In general, CSFuzz achieves a mean speedup in TTR of 97% against AFL, 58% against AFL++, 155% against FishFuzz, 87% against AFLGo, 91% against WindRanger, and 27% against CmpLog. Additionally, it achieves a mean speedup in TTE of 219% against AFL, 73% against AFL++, 166% against FishFuzz, 82% against AFLGo, 90% against WindRanger, and 65% against CmpLog. The average number of bugs discovered is 40% more than AFL, 10% more than AFL++, 28% more than FishFuzz, 16% more than AFLGo, 13% more than WindRanger, and 9% more than CmpLog.

The reason for the better performance of CSFuzz is that while other fuzzing techniques are not capable of capturing interesting states related to targets, they may fail to reach and trigger the corresponding vulnerabilities due to specific data conditions required. Unlike other fuzzers lacking additional feedback to guide the behavior of the PUT, capturing changes in relevant variable states helps reach and trigger correlated vulnerabilities.

C. Impact of Different Components (RQ2)

The methods of CSFuzz include critical variable identification, perception of new program states triggered by critical variables, corpus selection, and seed priority distinction. To investigate the impact of different components in CSFuzz, we disable each component individually and conduct experiments.

We use CSFuzz as the default setting and use TTE as the evaluation metric in the experiment. First, we disable the division of the adaptive state range of variables and use a uniform power of 2 for all variables to divide the interval as CSFuzz-T. Next, we use CSFuzz-C to represent the selection between the state corpus and code coverage corpus using a fixed probability. Then, we use CSFuzz-P to represent the absence of seed priority distinction within the state corpus. Additionally, we disable the feedback that triggers new program states, allowing the fuzzing to run solely on edge coverage feedback, denoted as CSFuzz-S. Finally, we also disable the distance information as CSFuzz-V, to evaluate the performance of vulnerability guidance disabled.

As shown in Table III, the default CSFuzz outperforms CSFuzz-T, CSFuzz-C, CSFuzz-P, CSFuzz-S, and CSFuzz-V in terms of TTE, improving 52%, 14%, 23%, 82%, 27% respectively. Experimental results show that CSFuzz-T lacks adaptive variable state range partitioning and is deficient in capturing program state. CSFuzz-C and CSFuzz-P indicate scheduling optimization, which improves the fuzzing performance. CSFuzz-S demonstrates that the identification and state detection of critical variables contributes to fuzzing. With disabled vulnerability guidance, the performance of CSFuzz-V decreased by 27%. While vulnerability guidance enhanced the overall effectiveness of crash reproduction, there was an 82% performance decline when critical variable states were disabled. This suggests that even without vulnerability guidance, critical variable states can still effectively facilitate DGF. Note that these components together make up CSFuzz, which proves the validity of monitoring the states of critical variables in DGF.

D. State Feedback Effect (RQ3)

The implementation of state feedback in CSFuzz requires additional instrumentation. To assess the execution speed overhead of CSFuzz, we compared it to the execution speed of AFL. This was done because our prototype is based on the implementation of AFL. We select seeds from the state corpus as inputs to ensure the execution of additional instrumentation code. Simultaneously, we employ the deterministic mode to guarantee identical test cases. As shown in Table IV, the integer numbers in the middle two columns represent the average execution counts per second, while the percentage values in the last column indicate the average performance overhead. The results show that the input execution speed introduced by CSFuzz is reduced by an average of 9%, indicating a small overhead. CSFuzz can bring objective efficiency gains with little performance overhead.

In addition to performance overhead, avoiding state explosion is crucial. During the fuzzing process, we maintain two corpora: a state queue that covers new program states and a

TABLE V
STATIC ANALYSIS AND STATE FEEDBACK RESULTS.

Project	v-nums	s-nums	e-nums
sqlite3	313	433	6099
lua	533	1158	3190
tiff	331	667	3708
pdftoppm	255	367	8489
xmllint	593	586	3392
sndfile	344	693	1712
asn1	163	302	2122

TABLE VI
PROGRAMS EVALUATED BY FUZZERS FOR DISCOVERING NEW BUGS

Program	Arguments	Program	Arguments
w3m	@@	podofopdfinfo	@@
xmllint	–valid –dtdattr –stream @@	ffmpeg	-i @@ test
exiv2	@@	freetype	@@
addr2line	-e @@	readelf	-w @@
strip	-o /dev/null @@	nm-new	-C @@
pdfalto	@@	lou_checktable	@@
nasm	-o /dev/null @@	objdump	-S @@
MP4Box	-dash 1000 -out /dev/null @@		

traditional edge coverage queue. In Table V, we present the number of critical variables identified through static analysis and the number of seeds stored in the different corpora after the 48 hours of the fuzzing campaign. The term "v-nums" represents the count of identified critical variables, "s-nums" represents the count of state corpus seeds, and "e-nums" represents the count of edge corpus seeds. On average, the proportion of seeds in the state corpus relative to the total corpus is 16%. In other words, the growth of the corpus in the fuzzing campaign is moderate and will not cause excessive overhead and state explosion. This is because we selected critical variables related to the target sites from the programs. And some are located deep within the program, which may result in them not being executed. Additionally, when test cases trigger new edges and new states simultaneously, the test cases are added only to the edge corpus and not to the state corpus. Finally, we partition the states of the critical variables into intervals to avoid state explosion. This is important for fuzzers because the seeds stored in the state corpus represent different parts of the input space due to changes in the runtime values of critical variables. Having an excessive number of seeds can lead to small differences between them, reducing the likelihood of discovering new bugs [39]. Moreover, an excessive number of seeds makes it challenging for fuzzers to maintain them.

E. Real Scenario Evaluation (RQ4)

To test whether CSFuzz helps in discovering previously unknown vulnerabilities, we applied CSFuzz, AFL, AFL++, FishFuzz, AFLgo, WindRanger, and CmpLog to the 15 latest versions of open-source software that are frequently tested by other fuzzing tools. The specific list of programs and their

```
GF_Err mpgviddmx_process(GF_Filter *filter)
{
...
while(remain)
{
  ...
  current = -1;
  if(ctx->bytes_in_header)
  {
    memmove(ctx->hdr_store + ctx->bytes_in_header,
    start, MIN_HDR_STORE - ctx->bytes_in_header);
    current = mpgviddmx_next_start_code(ctx->hdr_store,
    MIN_HDR_STORE);
  }
}
}
```

Listing 2. Simplified snippet of CVE-2024-32376

arguments is presented in Table VI. We collected a set of seeds that matched the target format as input. For DGF that requires targets, we set the recently patched or modified lines of code in the software as target sites. Under the same configuration, each fuzzer was executed five times, with each run lasting seven days. Ultimately, CSFuzz discovered four unique, previously unknown crashes, while the other fuzzers did not. After a brief analysis, we submitted these crashes to the developers. Three of them were quickly confirmed and fixed, with two CVEs (CVE-2024-32376 and CVE-2024-32377) assigned for gpac and libxml2, respectively.

We use one of the CVEs as an example to briefly explain how CSFuzz helps in triggering vulnerabilities. As shown in Listing 2, CVE-2024-32376 involves memory access violations caused by the use of malformed files. In the `remain` loop, `remain` indicates the number of bytes remaining to parse data, while `bytes_in_header` suggests potential bytes in `hdr_store`. When attempting to copy additional bytes with `memmove`, a heap buffer overflow occurs if `MIN_HDR_STORE - bytes_in_header > remain`. This code is deeply embedded in the program and subject to numerous constraints. As a result, other fuzzers fail to capture detailed feedback even when executed at the relevant position, missing the opportunity for comprehensive fuzzing. CSFuzz tracks changes in the values of `bytes_in_header` and `remain` variables, allocating resources to conduct thorough fuzzing around the target to trigger the vulnerability.

V. THREATS TO VALIDITY

In our experiments, we selected the hyperparameter settings for CSFuzz based on practical experience. Given that parameter tuning requires a significant investment of time and resources, and considering the satisfactory outcomes of our current experiments, we will leave improving the configurable options as part of our future work.

Furthermore, the state feedback mechanism in CSFuzz involves monitoring critical variables around the target sites. In cases where the target sites are difficult to access and lack nearby seed executions, the absence of relevant state feedback for these targets hinders CSFuzz from providing further valuable assistance.

VI. Related Work

Our study relates to the following fields of research:

Directed Grey-box Fuzzing. DGF is particularly effective for targeted crash reproduction and patch testing. AFLGo [15] was the first to introduce the concept of directed fuzzing. At runtime, AFLGo prioritizes seeds based on their harmonic mean distance to the targets. Hawkeye [19] initially proposed measuring the similarity between seed execution traces and target execution traces at the function level as a criterion for seed prioritization. BEACON [34] employs static analysis to terminate scenarios that cannot reach any targets. WindRanger [21] improves distance calculations from seeds to targets by considering basic blocks deviating from the target path. Fish-Fuzz [20] avoids inaccuracies in distance calculation by using inter-function distances for each seed. DAFL [40] performs data flow analysis to prioritize seeds based on the calculated distance according to data flow semantics.

These methods facilitate reaching the targets but do not consider additional feedback for thorough testing. In contrast, CSFuzz conducts more comprehensive testing of the targets by taking into account the states of critical variables, which aids in triggering target vulnerabilities. Furthermore, our approach is orthogonal to these methods, and we can adopt them to effectively facilitate reaching the areas near the targets.

Taint-analysis-based Fuzzing. Taint analysis is also widely used to enhance fuzzing. Angora [41] uses dynamic taint analysis to infer the shape of input bytes related to path constraints, determining where and how to mutate the inputs. REDQUEEN [35] observes the values used in comparison instructions and then colors the inputs using random mutations to infer taint related to direct copies of the inputs. GREYONE [32] employs dynamic taint analysis to guide the mutation process and prioritize seeds.

Although these methods guide mutations and help address path constraints, they primarily focus on the variables that influence those constraints, overlooking other variables. In contrast, we consider the state feedback of critical variables related to targets. Moreover, our approach can incorporate taint analysis techniques to enhance mutation, which does not conflict with existing methods.

Coverage-guided Grey-box Fuzzing. Edge coverage is a form of coverage feedback that can effectively guide fuzzing. Recent work has proposed enhancing edge coverage using context-sensitive edge coverage [41] or path coverage [39], [42]. Moreover, there have been methods that explore using data information as a complement. INVSCOV [43] collects program invariants through pre-execution and incorporates invariant coverage feedback. DDFuzz [44] and DATAFLOW [45] introduce data flow edges to explore coverage feedback beyond traditional control flow edges.

However, simply increasing coverage signals means that more inputs are considered valuable, which can slow down the fuzzer due to indiscriminate testing. Although Ijon [46] improves coverage metrics by using important annotations marked in the program, it requires experts to have a deep understanding of the program to manually label them. In contrast, CSFuzz moves the problem statement to directed fuzzing, focusing on state coverage feedback from more critical variables. This approach avoids state explosion and balances effectiveness with efficiency, leading to more effective bug discovery.

VII. Conclusion

In this paper, we propose the concept of monitoring the states of critical variables. We implement a prototype of CSFuzz to facilitate directed greybox fuzzing. It identifies critical variables associated with the target sites and preserves the seeds that trigger new states at runtime. We perform experimental evaluations of CSFuzz on benchmarks and real-world programs. The evaluation results show that CSFuzz can expose vulnerabilities faster than state-of-the-art fuzzers. Notably, CSFuzz demonstrates practical usefulness by discovering two 0-day vulnerabilities in two extensively fuzzed programs.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and opinions for improving this work. We also thank Xiaobei Han, Zhijie Zhang, and Haolai Wei for their discussion about this work. This work is partially supported by the National Natural Science Foundation of China (No. 62172407), and the Youth Innovation Promotion Association CAS.

References

[1] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[2] G. Inc, "Syzkaller-kernel fuzzer," 2016, accessed July 22, 2024. [Online]. Available: https://github.com/google/syzkaller

[3] K. Serebryany, "Libfuzzer: A library for coverage-guided fuzztesting (within llvm)," 2017, accessed July 22, 2024. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[4] M. Zalewski, "American fuzzy lop," 2014, accessed July 22, 2024. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[5] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 679–696.

[6] 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*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 475–485.

[7] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966.

[8] 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 *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 769–786.

[9] K. Serebryany, "Oss-fuzz - google's continuous fuzzing service for open source software." Vancouver, BC: USENIX Association, Aug. 2017.

[10] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 511–522.

[11] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 474–484.

[12] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 206–215.

[13] 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*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 543–553.

[14] 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*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1032–1043.

[15] 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*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 2329–2344.

[16] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 213–223.

[17] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 144–155.

[18] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: Let existing test cases help," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 910–913.

[19] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 2095–2108.

[20] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1343–1360.

[21] Z. Du, Y. Li, Y. Liu, and B. Mao, "WindRanger: A directed greybox fuzzer driven by deviation basic blocks," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 2440–2451.

[22] H. Huang, P. Yao, H. CHIU, Y. Guo, and C. Zhang, "Titan: Efficient multi-target directed greybox fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 62–62.

[23] H. Rong, W. You, X. Wang, and T. Mao, "Toward Unbiased Multiple-Target Fuzzing with Path Diversity," Oct. 2023.

[24] S. Inc, "Heartbleed," 2020, accessed July 22, 2024. [Online]. Available: https://heartbleed.com/

[25] Taviso, "Cloudbleed," 2017, accessed July 22, 2024. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1139

[26] K. Serebryany, "A memory error detector," 2012, accessed July 22, 2024. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[27] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2777–2794.

[28] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Nov. 2020.

[29] Gpac, "An open-source multimedia framework," 2000, accessed July 22, 2024. [Online]. Available: https://github.com/gpac/gpac

[30] Libxml2, "GNOME project," 1999, accessed July 22, 2024. [Online]. Available: https://github.com/GNOME/libxml2/

[31] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[32] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2577–2594.

[33] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761.

[34] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: Directed grey-box fuzzing with provable path pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 36–50.

[35] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:85546717

[36] M. Corporation, "Mitre cve database," 1958, accessed July 22, 2024. [Online]. Available: https://cve.mitre.org/

[37] A. Hazimeh, "Magma artifacts," 2020, accessed July 22, 2024. [Online]. Available: https://osf.io/resj8/

[38] Docker, "Use containers to build, share and run your applications." 2021, accessed July 22, 2024. [Online]. Available: https://www.docker.com/resources/what-container

[39] 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)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15.

[40] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "DAFL: Directed grey-box fuzzing guided by data dependency," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4931–4948. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

[41] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[42] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "PathAFL: Path-coverage assisted fuzzing," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 598–609.

[43] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The Use of Likely Invariants as Feedback for Fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2829–2846.

[44] A. Mantovani, A. Fioraldi, and D. Balzarotti, "Fuzzing with data dependency information," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroSP)*, 2022, pp. 286–302.

[45] A. Herrera, M. Payer, and A. L. Hosking, "Dataflow: Toward a data-flow-guided fuzzer," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, Jul. 2023. [Online]. Available: https://doi.org/10.1145/3587156

[46] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring Deep State Spaces via Fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1597–1612.