electronics

MDPI

Article

REFUZZ: A Remedy for Saturation in Coverage-Guided Fuzzing

Qian Lyu [1], Dalin Zhang [1,*], Rihan Da [1] and Hailong Zhang [2,*]

[1] School of Software Engineering, Beijing Jiaotong University, Beijing 100044, China ; 20126318@bjtu.edu.cn (Q.L.); 20126291@bjtu.edu.cn (R.D.)
[2] Department of Computer and Information Sciences, Fordham University, Bronx, NY 10458, USA
* Correspondence: dalin@bjtu.edu.cn (D.Z.); hzhang285@fordham.edu (H.Z.)

Abstract: Coverage-guided greybox fuzzing aims at generating random test inputs to trigger vulnerabilities in target programs while achieving high code coverage. In the process, the scale of testing gradually becomes larger and more complex, and eventually, the fuzzer runs into a saturation state where new vulnerabilities are hard to find. In this paper, we propose a fuzzer, REFUZZ, that acts as a complement to existing coverage-guided fuzzers and a remedy for saturation. This approach facilitates the generation of inputs that lead only to covered paths by omitting all other inputs, which is exactly the opposite of what existing fuzzers do. REFUZZ takes the test inputs generated from the regular saturated fuzzing process and continue to explore the target program with the goal of *preserving* the code coverage. The insight is that coverage-guided fuzzers tend to underplay already covered execution paths during fuzzing when seeking to reach new paths, causing covered paths to be examined insufficiently. In our experiments, REFUZZ discovered tens of new unique crashes that AFL failed to find, of which nine vulnerabilities were submitted and accepted to the CVE database.

Keywords: remedial testing; greybox fuzzing; vulnerability detection; enhanced security

Citation: Lyu, Q.; Zhang, D.; Da, R.; Zhang, H. REFUZZ: A Remedy for Saturation in Coverage-Guided Fuzzing. *Electronics* **2021**, *10*, 1921. https://doi.org/10.3390/electronics10161921

Academic Editor: Arman Sargolzaei

Received: 21 June 2021
Accepted: 8 August 2021
Published: 10 August 2021

1. Introduction

Software vulnerabilities are regarded as a significant threat in information security. Programming languages without a memory reclamation mechanism (such as C/C++) have the risk of memory leaks, which may expose irreparable risks [1]. With the increase in software complexity, it is impractical to reveal all abnormal software behaviors manually. Fuzz testing, or *fuzzing*, is a (semi-) automated technology to facilitate software testing. A fuzzing tool, or *fuzzer*, feeds random inputs to a target program and, meanwhile, monitors unexpected behaviors during software execution to detect vulnerabilities [2]. Among all fuzzers, coverage-guided greybox fuzzers (CGF) have become one of the most popular ones due to their high deployability and scalability, e.g., AFL [3] and LibFuzzer [4]. They have been successfully applied in practice to detect thousands of security vulnerabilities in open-source projects [5].

Coverage-guided greybox fuzzing relies on the assumption that *more run-time bugs could be revealed if more program code is executed*. To find bugs as quickly as possible, AFL and other CGFs try to maximize the code coverage. This is because a bug at a specific program location can only be triggered unless that location is covered by some test inputs. A CGF utilizes light-weight program transformation and dynamic program profiling to collect run-time coverage information. For example, AFL instruments the target program to record transitions at the basic block level. The actual fuzzing process starts with an initial corpus of seed inputs provided by users. AFL generates a new set of test inputs by randomly mutating the seeds (such as bit flipping). It then executes the program using the mutated inputs and records those that cover new execution paths. AFL continually repeats this process, but starts with the mutated inputs instead of user-provided seed inputs. If there are any program crashes and hangs, for example, caused by memory errors, AFL would also report the corresponding inputs for further analysis.

When a CGF is applied, the fuzzing process does not terminate automatically. Practically, users need to decide when to end this process. In a typical scenario, a user sets a timer for each CGF run and the CGF stops right away when the timer expires. However, researchers have discovered empirically that, within a fixed time budget, exponentially more machines are needed to discover each new vulnerability [6]. With a limited number of machines, the CGF could rapidly reach a *saturation* state in which, by continuing the fuzzing, it is difficult to find new unique crashes (where exponentially more time is needed). Then, *what can we do to improve the capability of CGF to find bugs with constraints on time and CPU power*? In this work, we try to provide one solution to this question.

Existing CGFs are biased toward test inputs that can explore new program execution paths. These inputs are prioritized in subsequent mutations. Inputs that do not discover new coverage are considered unimportant and are not selected for mutation. However, in practice, this extensive coverage-guided path exploration may *hinder the discovery of or even overlook potential vulnerabilities on specific paths*. The rationale is that an execution path in one successful run may not be bug-free in all runs. Simply dumping "bad" inputs may cause insufficient testing of their corresponding execution paths. Rather, special attention should be paid to such inputs and paths. Intuitively, an input covering a path is more likely to cover the same path after mutation than any other arbitrary inputs. Although an input cannot trigger, in one execution, the bug in its path, it is possible that the input can do so after a few fine-grained mutations. In short, by focusing on the new execution paths, the CGFs can discover an amount of vulnerabilities in a fixed time, but they also omit some vulnerabilities, which need to be repeatedly tested on the specific execution path multiple times to be found.

Based on this, we propose a lightweight extension of CGF, REFUZZ, that can effectively find tens of new crashes within a fixed amount of time on the same machines. The goal of REFUZZ is not to achieve as high code coverage as possible. Instead, it aims to *detect new unique crashes on already-covered execution paths in a limited time*. In REFUZZ, test inputs that do not explore new paths are regarded as favored. They are prioritized and mutated often to examine the same set of paths repeatedly. All other mutated inputs are omitted from execution. As a prototype, we implement REFUZZ on top of AFL. In our experiments, it successfully triggered 37, 59, and 54 new crashes in our benchmarks that were not found by AFL, using three different experimental settings, respectively. Finally, we discovered nine vulnerabilities accepted to the CVE database.

In particular, REFUZZ incorporates two stages. Firstly, in the *initial* stage, AFL is applied as usual to test the target program. The output of this stage is a set of crash reports and a corpus of mutated inputs used during fuzzing. In addition, we record the code coverage of this corpus. Secondly, in the *exploration* stage, we use the corpus and coverage from the previous stage as seed inputs and initial coverage, respectively. During the testing process, instead of rewarding inputs that cover new paths, REFUZZ only records and mutates those that converge to the initial coverage, i.e., they contribute no new coverage. To further improve the performance, we also review the validity of each mutated input before execution and promote non-deterministic mutations, if necessary. In practice, the second stage may last until the fuzzing process becomes saturated.

Note that REFUZZ is not designed to replace CGF but as a *complement* and a *remedy* for saturation during fuzzing. In fact, the original unmodified AFL is used in the initial stage. The objective of the exploration stage is to verify whether new crashes can be found on execution paths that have already been covered by AFL and whether AFL and CGFs, in general, miss potential vulnerabilities on these paths while seeking to maximize code coverage.

We make the following contributions.

- We propose an innovative idea in which, though the input cannot trigger a bug over one execution time, it is possible that the input can do so after a few fine-grained mutations.

- We propose a lightweight extension of CGF, ReFuzz, that can effectively find tens of new crashes within a fixed amount of time on the same machines.
- We develop various supporting techniques, such as reviewing the validity of each mutated input before execution, and promote non-deterministic mutations if necessary to further improve the performance.
- We propose a new mutation strategy on top of AFL. If the input does not cover a new execution path, it is regarded as valuable, which will help to cover a specific execution path over multiple times.
- We evaluate ReFuzzon four real-world programs collected from prior related work [7]. It successfully triggered 37, 59 and 54 new unique crashes in the three different experimental configurations and discovered nine vulnerabilities accepted to the CVE database.

The rest of the paper is organized as follows. Section 2 introduces fuzzing and AFL, as well as a motivating example to illustrate CGFs mutating strategy limitations. Section 3 describes the design details of ReFuzz. We report the experimental results and discussion in Sections 4 and 5. Section 6 discusses the related work, and finally, Section 7 concludes our work.

2. Background

2.1. Fuzzing and AFL

Fuzzing is a process of automatic test generation and execution with the goal of finding bugs. Over the past two decades, security researchers and engineers have proposed a variety of fuzzing techniques and developed a rich set of tools that helped to find thousands of vulnerabilities (or more) [8]. Blackbox fuzzing randomly mutates test inputs and examines target programs with these inputs. Whitebox fuzzing, on the other hand, utilizes advanced, sophisticated program analyses, e.g., symbolic execution [9], to systematically exercise all possible program execution paths. Greybox fuzzing sits in between the former two techniques. The testing is guided by run-time information gathered from program execution. Due to its high scalability and ease of deployment, coverage-guided greybox fuzzing gains popularity in both the research community and industry. Specifically, AFL [3] and its derivations [10–14] have received plenty of attention.

Algorithm 1 shows the skeleton of the original AFL algorithm. (The algorithm does not distinguish between deterministic and non-deterministic—totally random mutations for simplicity.) Given a program under test and a set of initial test inputs (i.e., the seeds), AFL instruments each basic block of the program to collect *block transitions* during the program execution and runs the program with mutated inputs derived from the seeds. The generation of new test inputs is guided by the collected run-time information. More specifically, if an input contributes no crash or new coverage, it is regarded as useless and is discarded. On the other hand, if it covers new state transitions, it is added as a new entry in the queue to produce new inputs since the likelihood of these resulting inputs achieving new coverage is heuristically higher, compared to other arbitrary inputs. However, this coverage-based exploration strategy leads to strong bias toward such inputs, making already explored paths probabilistically less inspected. In our experiments, we found that these paths actually contained a substantial number of vulnerabilities, causing programs to crash.

AFL mutates an input at both a coarse-grained level, which incorporates the changing bulks of bytes, and a fine-grained level, which involves byte-level modifications, insertions and deletions [15]. In addition, AFL adopts two strategies to apply the mutation, i.e., deterministic mutation and random mutation. In fuzzing, AFL maintains a seed queue that stores the *initial* test seeds provided by users and new test cases screened by the fuzzer. For one input in the seed queue, which has applied deterministic mutations, it will no longer be mutated through deterministic mutation in subsequent fuzzing. The deterministic mutation, including bitflip, arithmetic, interest, and dictionary methods, is one in which a new input is obtained by modifying the content of the input at a specific byte position and

every input is mutated in the same way. In particular, during the interest and dictionary mutation stages, some special contents and tokens automatically generated or provided by users are replaced or inserted into the original input. On the contrary, the havoc and splice called random mutations would always be applied until the fuzzing stops. In the havoc stage, a random number is generated as the mutation combination number. According to the number, one random mutation method is selected each time, and then applied to the file in turn. In the next stage, called splice, a new input is produced by splicing two seed inputs, and the havoc mutation is continued on the file.

Algorithm 1: OriginalAFL

Input: The target program P; the initial set of seed inputs *initSeeds*.

```
1  queue ← initSeeds
2  crashes ← ∅
3  while in fuzzing loop do
4      foreach input ∈ queue do
5          foreach mutation ∈ allMutations do
6              newInput ← MUTATE(input, mutation)
7              result ← RUN(P, newInput)
8              if CRASH(result) then
9                  crashes ← crashes ∪ {result}
10             else if NEWCOVERAGE(result) then
11                 queue ← queue ∪ {newInput}
12             end
13         end
14     end
15 end
16 return queue, crashes
```

Note that AFL is unaware of the structure of inputs. For example, it is possible that a MP3 file is generated from a PDF file because the magic number is changed by AFL. It is inefficient to test a PDF reader with a MP3 file since the execution will presumably terminate early, as the PDF parser does not accept non-PDF files, causing the major components not to be tested. Our implementation of ReFuzz tackles this problem by adding an extra check of validness of newly generated test inputs, as discussed in Section 3.

2.2. Motivating Example

Figure 1a shows a code snippet derived from the `pdffonts` program, which analyzes and lists the fonts used in a Portable Document Format (PDF) file. Class `Dict` defined at line 1–10 stores an array of entries. Developers can call the `find` function defined at line 12 to retrieve the corresponding entry by a key. In the experiments, we test this program by running both AFL and ReFuzz with the AddressSanitizer [16] to detect memory errors. Figure 1b shows the crashing trace caused by a heap buffer overflow error found only by ReFuzz. The crash is caused by accessing the `entries` array during the iteration at line 14–17 in Figure 1a. The root cause of this error is inappropriate destruction of the dictionary in the `XRef` and `Object` classes when `pdffonts` attempts to reconstruct the *cross-reference table* (xref for short, which internally uses a dictionary) for locating objects in the PDF file, e.g., bookmarks and annotations. The crash is triggered when the xref table of the test input is mostly valid (including the most important entries, such as "Root", "size", "Info", and "ID") but cannot pass the extra check to investigate whether the PDF file is encrypted. When the program issues a search of key "Encrypt", the dictionary has already been destructed by a previous query that checks for the validness of the xref table. A correct implementation should make a copy of the dictionary after the initial check.

```
1  class Dict {
2  public:
3    ...
4  private:
5    XRef *xref; // the xref table for this PDF file
6    DictEntry *entries; //array of entries
7    int length; //number of entries in dictionary
8    ...
9    DictEntry *find(char *key);
10 };
11 ...
12 inline DictEntry *Dict::find(char *key){
13   int i;
14   for (i = 0; i < length; ++i) {
15     if (!strcmp(key, entries[i].key))
16       return &entries[i];
17   }
18   return NULL;
19 }
20 ...
```

(**a**) Code derived from pdffonts

Function	File and Line
`main`	/Xpdf/pdffonts.cc:117
`PDFDoc::PDFDoc`	/Xpdf/PDFDoc.cc:96
`PDFDoc::setup`	/Xpdf/PDFDoc.cc:120
`XRef::XRef`	/Xpdf/XRef.cc:107
`XRef::checkEncrypted`	/Xpdf/XRef.cc:459
`Object::dictLookup`	/Xpdf/Object.h:252
`Dict::lookup`	/Xpdf/Dict.cc:72
`Dict::find`	/Xpdf/Dict.cc:56

(**b**) The crashing trace caused by a heap buffer overflow

Figure 1. The motivating example.

It is relatively expensive to find this vulnerability using AFL, compared to ReFuzz. In our experiments, by running AFL for 80 h, AFL failed to trigger this bug, even with the help of the AddressSanitizer tool. The major reason is that the check for validness of xref and the check for encryption of the PDF file are the first step when pdffonts parses an arbitrary file—that is, they are presumably regarded as "old" paths for most cases. When using AFL, if a test input does not cover a new execution path, the chance of mutating this input is low. In other words, the execution path covered by the input is less likely to be covered again (or is covered but by less "interesting" inputs) and the examination of the the two checks might not be enough to reveal subtle bugs, such as the one in Figure 1b.

To tackle this problem, ReFuzz does not aim at high code coverage. On the contrary, we want to detect new vulnerabilities residing in covered paths and to verify that AFL ignores possible crashes in such paths while paying attention to coverage. ReFuzz utilizes the corpus obtained in the initial stage (which runs the original AFL) as the seeds for the exploration stage. It only generates test inputs that linger on the execution paths that are covered in the first stage but not investigated sufficiently. In the next section, we provide more details about the design of ReFuzz.

3. Design of ReFuzz

3.1. Overview

We propose ReFuzz to further test the program under test with inputs generated by AFL to trigger unique crashes that were missed by AFL. ReFuzz consists of two stages, i.e., the *initial* stage and the *exploration* stage. In the initial stage, the original AFL is applied. The initial seed inputs are provided by the user. The output is an updated seed queue, including both the seed inputs and the test inputs covered new execution paths during fuzzing. In the exploration stage, ReFuzz uses this queue as the initial seed input, applying

a novel mutation strategy designed for investigating previously executed paths to generate new test inputs. Moreover, only inputs that passed the extra format check are added to the seed queue and participate in subsequent mutations and testing. Figure 2 shows the workflow of ReFuzz.

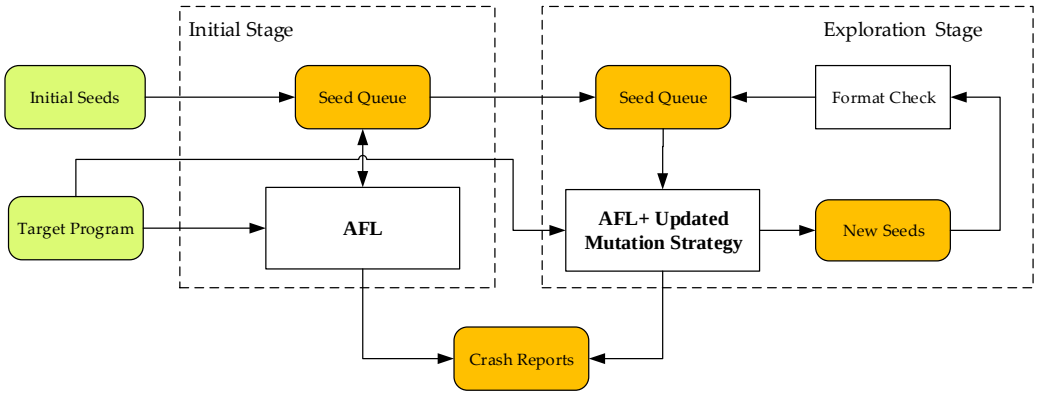

Figure 2. ReFuzz overview.

Algorithm 2 depicts the algorithmic sketch of ReFuzz. (Our implementation skips duplicate deterministic mutations of inputs in the Mutate function.) The highlighted lines are new, compared to the original AFL algorithm. The ReFuzz algorithm takes two additional parameters besides *P* and *initSeeds*: *et*, the time allowed for the initial stage, and *ct*, the time limit for performing deterministic mutations. We discuss *ct* in the next subsection. At line 6 in Algorithm 2, when the elapsed time is less than *et*, ReFuzz is in the initial stage, and the original AFL algorithm is applied. When the elapsed time is greater than or equal to *et* (line 8–24), the testing enters the exploration stage. ReFuzz uses in this stage the input corpus queue obtained in the initial stage and applies a novel mutation strategy to generate new test inputs. If a new input passes the format check, it would be fed to the target program. The input that preserved the code coverage (i.e., did not trigger new paths) would be added to the queue. In the experiments, we set *et* to various values to evaluate the effectiveness of ReFuzz under different settings.

3.2. Mutation Strategy in Exploration Stage

ReFuzz adopts the same set of mutation operators as in AFL, including bitflip, arithmetic, value overwrite, injection of dictionary terms, havoc, and splice. The first four methods are *deterministic* because of their slight destructiveness to the seed inputs. The latter two methods will significantly damage the structure of an input, which are *totally random*. To facilitate the discovery of crashes, as shown in Algorithm 2, we introduce a parameter *ct* to limit the time since the last crash during the fuzzing process for deterministic mutations. If an input is undergoing deterministic mutation operations and no new crashes are found for a long time (>*ct*), ReFuzz will skip the current mutation operation and perform the next random mutation (line 11 of Algorithm 2). In the experiments, we initialize *ct* to 60 min and set it incrementally for each deterministic mutation. Specifically, the *n*-th deterministic mutation is skipped if there no crash is triggered in the past *n* hours by mutating an input. ReFuzz will try other more destructive mutations to facilitate the efficiency of fuzzing.

As introduced in Section 1, ReFuzz does not aim at high code coverage. Instead, it generates inputs that converge to the existing execution paths. During the initial stage, AFL saves the test inputs that have explored new execution paths in the input queue. An execution path consists of a series of *tuples*, where each tuple records the run-time transition between two basic blocks in the program code. A path is new when the input results in (1) the generation of new tuples or (2) changing of the *hit count* (i.e., the frequency) of an existing tuple. Instead, the PreserveCoverage function in Algorithm 2 checks

whether new tuples are covered and returns false if this is the case. It returns true if any hit count along a path is updated. We add test inputs that preserves the coverage into the queue to participate in the next round of mutation as seeds. Using this mutation strategy, ReFuzz can effectively attack specific code areas that have been covered but are not well-tested and find vulnerabilities.

Algorithm 2: ReFuzz

Input: The target program P; the initial set of seed inputs *initSeeds*; the time to enter the exploration stage *et*; the time limit for performing deterministic mutations *ct*.

```
1  queue ← initSeeds
2  crashes ← ∅
3  lastCrash ← 0
4  while in fuzzing loop do
5      if elapsedTime < et then                                  // Initial stage
6          queue, crashes ← OriginalAFL(P, queue)
7      else                                                      // Exploration stage
8          foreach input ∈ queue do
9              foreach mut ∈ allMutations do
10                 if ct < elapsedTime − lastCrash ∧ IsDeterministic(mut) then
11                     continue
12                 end
13                 newInput ← Mutate(input, mut)
14                 if FormatCheck(newInput) then
15                     result ← Run(P, newInput)
16                     if Crash(result) then
17                         crashes ← crashes ∪ {result}
18                         lastCrash ← elapsedTime
19                     else if PreserveCoverage(result) then
20                         queue ← queue ∪ {newInput}
21                     end
22                 end
23             end
24         end
25     end
26 end
27 return queue, crashes
```

3.3. Input Format Checking

Blindly feeding random test inputs to the target program leads to low performance of the fuzzer since they are likely to fail the initial input validation [8]. For instance, it is better to run a audio processing program with a MP3 file instead of an arbitrary file. Since AFL is unaware of the expected input format for each program under test, it is usual that the structure of an input is changed by random mutation operations. We propose to add an extra, light-weight format check before each program run to reduce the unnecessary overhead caused by invalid test inputs. As an exemplar, in the experiments, we check whether each input is a PDF file when testing a PDF reader and discard those that do not conform to the PDF format during testing. Specifically, in our implementation, ReFuzz takes an extra command-line argument, indicating the expected format of inputs. For each mutated input, ReFuzz checks the magic number of each input file and only adds it to the queue for further mutation if it passes the check.

4. Evaluation

4.1. Experimental Setup

To empirically evaluate the REFUZZ and its performance in finding vulnerabilities, we implement REFUZZ on top of AFL and conduct experiments on a Ubuntu V16.04.6 LTS machine with 16-core Xeon E7 2.10 GHz CPU and 32 GB RAM, using 4 programs that were also used by prior, related work [7] . Table 1 shows the details of the subjects used in our experiments. Columns "Program" and "Version" show the program names and versions. Columns "#Files" and "#LOC" list the number of files and lines of code in each program, respectively.

Table 1. Experimental subjects.

Program	Version	#Files	#LOC
pdftotext	xpdf-2.00	133	51,399
pdftopbm	xpdf-2.00	133	51,399
pdffonts	xpdf-2.00	133	51,399
MP3Gain	1.5.2	39	8701

4.2. Vulnerability Discovery

A crucial factor in evaluating a fuzzer's performance is its ability to detect vulnerabilities. We configure REFUZZ to run three different experiments for 80 h with identical initial corpus by modifying et al. Table 2 describes the time for the initial stage and the exploration stage. In the first stage, the original AFL is applied without the additional test input format checking. Then, REFUZZ takes the input queue as the initial corpus for the second stage and uses an extra parameter to pass the expected input type to the target program, e.g., PDF.

Table 2. Experimental setup.

#	Init Time	Expl Time	Total
1	60 h	20 h	80 h
2	50 h	30 h	80 h
3	40 h	40 h	80 h
4	80 h	0 h	80 h

During the fuzzing process, the fuzzer records the information of each program crash along with the input that caused the crash. To avoid duplicates in the results, we use the *afl-cmin* [17] tool in the AFL toolset to minimize the final reports by eliminating redundant crashes and inputs. Tables 3–5 show the statistics of unique crashes triggered by REFUZZ. Note that the numbers in column "Init+Expl" are not exactly the sum of the numbers in columns "Init" and "Expl". This is because REFUZZ discovers duplicate crashes in the initial stage and the exploration stage. Additionally, the numbers in column "New" are discovered by REFUZZ but not AFL. After applying afl-cmin, only the unique crashes are reported.

We also run AFL for 80 h and report the number of crashes in Table 6. The total number in column "Init" is less than the number in column "Init+Expl" in Tables 3 and 5. This indicates that REFUZZ can find more unique crashes within 80 h. In Table 4, the data in column "Init" are much fewer than the other two experimental configurations, so they are fewer than the total number of crashes in Table 6. As described in Table 7, we compare the average and variance data of the unique crashes obtained though the four programs under three different experimental configurations. The data in column "Variance" have a large deviation, which reflects the randomness of the fuzzing.

From the Tables 3–5, we can see that new unique crashes are detected during the exploration stage in all three experimental settings, except for pdftopbm, which has 0 new

crashes, shown in Table 4. By applying the novel mutation strategy in the exploration stage and input format checking, REFUZZ discovers 37, 59, and 54 new unique crashes that are not discovered by AFL. These crashes are hard to find if we simply focus on achieving high code coverage since they reside in already covered paths and are not examined sufficiently with various inputs in that some vulnerabilities are detected by relying on plenty of special-type inputs.

Table 3. Number of unique crashes (60 + 20).

Program	REFUZZ			New
	Init	Expl	Init+Expl	
pdftotext	29	4	30	1
pdftopbm	33	14	40	7
pdffonts	154	74	164	10
MP3Gain	92	55	111	19
Total	308	147	345	37

Table 4. Number of unique crashes (50 + 30).

Program	REFUZZ			New
	Init	Expl	Init+Expl	
pdftotext	11	9	18	7
pdftopbm	8	1	8	0
pdffonts	153	81	164	11
MP3Gain	74	76	115	41
Total	246	167	305	59

Table 5. Number of unique crashes (40 + 40).

Program	REFUZZ			New
	Init	Expl	Init+Expl	
pdftotext	22	6	25	3
pdftopbm	32	25	41	9
pdffonts	148	88	164	16
MP3Gain	101	61	127	26
Total	303	180	357	54

Table 6. Number of unique crashes (80 + 0).

Program	Init
pdftotext	35
pdftopbm	39
pdffonts	171
MP3Gain	96
Total	341

Table 7. Average and variance of unique crashes.

Program	60 + 20	50 + 30	40 + 40	Average	Variance
pdftotext	30	18	25	24.3	24.2
pdftopbm	40	8	41	29.7	234.9
pdffonts	164	164	164	164	0
MP3Gain	111	115	127	117.7	46.2

Figure 3 shows the proportion of newly discovered unique crashes among all crashes that are triggered by REFUZZ in the exploration stage. For example, for pdftotext, the number of new unique crashes is greater than half of the total number of unique crashes (in the "40 + 40" setting). We can see that by preserving the code coverage and examining covered execution paths more, we can discover a relatively large number of new vulnerabilities that might be neglected by regular CGF, such as AFL. Note that this does not mean that AFL and others cannot find such vulnerabilities. It just implies that they have a lower chance of finding the vulnerabilities within a fixed amount of time, while REFUZZ is more likely to trigger these vulnerabilities, given the same amount of time.

In addition, we set up 12 extra experiments. The corpus obtained by running AFL for 80 h is used as the initial input of the exploration stage; then, the target programs are tested by REFUZZ for 16 h. The purpose is to verify whether REFUZZ can always find new unique crashes when AFL is saturated. The experimental data are recorded in Table 8. The column "Number of experiments" records the number of new unique crashes found by REFUZZ in 12 experiments. It can be proved that when the same initial inputs are provided, REFUZZ can always find new crashes that are not repeated with AFL, even though the fuzzing is random.

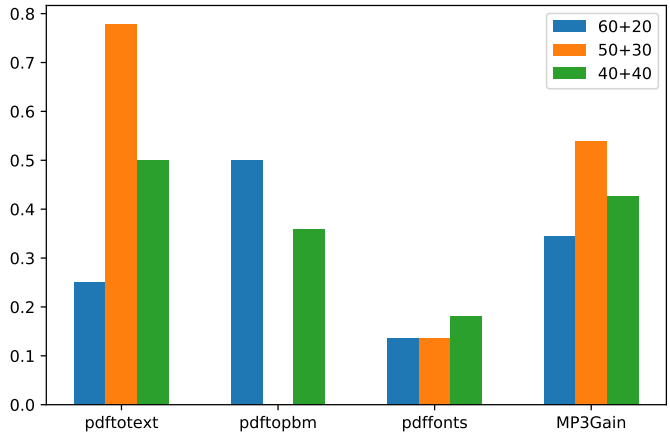

Figure 3. Proportion of newly discovered unique crashes in the exploration stage of REFUZZ.

Table 8. Number of new unique crashes (80 + 16).

Program	Number of Experiments												Average
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
pdftotext	1	7	3	2	4	4	0	0	3	5	7	4	3.3
pdftopbm	3	6	9	2	2	5	5	2	3	4	8	3	4.3
pdffonts	18	15	19	11	8	13	11	18	19	19	7	22	15
MP3Gain	1	6	15	8	7	13	7	7	8	22	14	8	9.7

We have submitted our findings in the target programs to the CVE database. Table 9 shows a summary of nine new vulnerabilities that were found by REFUZZ in our experiments. We are working on analyzing the rest crashes and will release more details in the future.

4.3. Code Coverage

As described earlier, the goal of REFUZZ is to test whether new and unique crashes can be discovered on covered paths after regular fuzzing in a limited time, instead of aiming at high code coverage. We collected the code coverage information during the execution of REFUZZ and found that the coverage for each target program remained the same during the exploration stage, which is to be expected. The results also show that

AFL only achieved slightly higher coverage compared to ReFuzz in the exploration stage, which implies that AFL ran into a saturation state, which signifies a demand for new strategies to circumvent such scenarios. ReFuzz is one such remedy, and our experimental results show its effectiveness in finding new crashes.

Table 9. Submitted vulnerabilities.

ID	Description
CVE-2020-25007	double-free vulnerability that may allow an attacker to execute arbitrary code
CVE-2020-27803	double-free on a certain position in thread
CVE-2020-27805	heap buffer access overflow in XRef::constructXRef() in XRef.cc
CVE-2020-27806	SIGSEGV in function scanFont in pdffonts.cc
CVE-2020-27807	heap-buffer-overflow in function Dict::find(char *) in Dict.cc
CVE-2020-27808	heap-buffer-overflow in function Object::fetch(XRef *, Object *) in Object.cc
CVE-2020-27809	SIGSEGV in function XRef::getStreamEnd(unsigned int, unsigned int *) in XRef.cc
CVE-2020-27810	heap-buffer-overflow in function Dict::find(char *) in Dict.cc
CVE-2020-27811	SIGSEGV in function XRef::readXRef(unsigned int *) in XRef.cc

5. Discussion

ReFuzz is effective at finding new unique crashes that are hard to discover using AFL. This is because some execution paths need to be examined multiple times with different inputs to find hidden vulnerabilities. The coverage-first strategy in AFL and other CGFs tends to overlook executed paths, which may hinder further investigation of such paths. However, such questions as "when should we stop the initial stage in ReFuzz and enter the exploration stage to start the examination of these paths", and "how long should we spend in the exploration stage of ReFuzz" remain unanswered.

How long should the initial stage take? As described in Section 4, we performed three different experiments with *et* set to 60, 50, and 40 h to gather empirical results. The intuition is that the effect of using the original AFL to find bugs would be the best when *et* is 60 h since it is to be expected that more paths could be covered and more unique crashes could be triggered if we apply the fuzzer for a longer time in the initial stage. However, our experimental results in Tables 3–5 show that the fuzzing process is unpredictable. The total number of unique crashes triggered in the initial stage of 60 h is close to 40 h (308 vs. 303), while the number obtained in 50 h is less than that of 40 h (246 vs. 303). In Algorithm 2, as well as our implementation of the algorithm, we allow the user to decide when to stop the initial stage and set *et* based on their experience and experiments. Generally, regarding the appropriate length of the initial stage, we suggest that users should pay attention to the dynamic data in the fuzzer dashboard. The code coverage remains stable, the color of the cycle numbers (*cycles done*) transforms from purple to green, or the last discovered unique crashes (*last uniq crash time*) have passed a long time, which indicates that continuing to test will not bring new discoveries. The best rigorous method is to combine these pieces of reference information to determine whether the initial stage should be paused.

How long should the exploration stage take? We conducted an extra experiment using ReFuzz with the corpus obtained from the 80-h run of AFL. We ran ReFuzz for 16 h and recorded the number of unique crashes per hour. In the experiment, each program was executed with ReFuzz for 12 trials. The raw results are shown in Figure 4 and the mean of the 12 trials are shown in Figure 5. In both figures, the *x*-axes show the number of bugs (i.e., unique crashes) and the y-axes show the execution time in hours. We can see that given a fixed corpus of seed inputs, the performance of ReFuzz in the exploration stage varies a lot in the 12 trials. This is due to the nature of random mutations. Overall, we can see from the figures that in the exploration stage, ReFuzz follows the empirical rule that finding a new vulnerability requires exponentially more time [6]. However, this does not negate the effectiveness of ReFuzz in finding new crashes. We suggest that the best test

time to terminate the remedial testing is still when the exploration reaches saturation, and the relevant guidelines at the initial stage can be considered here.

Is REFUZZ effective as remedy for CGF? Many researchers have proposed remedial measures to CGFs. Driller [18] combines fuzzing and symbolic execution. When a fuzzer becomes stuck, symbolic execution can calculate the valid input to explore deeper bugs. T-Fuzz [19] detects whenever a baseline mutational fuzzer becomes stuck and no longer produces inputs that extend the coverage. Then, it produces inputs that trigger deep program paths and, therefore, find vulnerabilities (hidden bugs) in the program. The main cause of the saturation is due to the fact that AFL and other CGFs strongly rely on random mutation to generate new inputs to reach more execution paths. Our experimental results suggest that new unique crashes can actually be discovered if we leave code coverage aside and continue to examine the already covered execution paths by applying mutations (as shown in Tables 3–5). They also show that it is feasible and effective to use our approach as a remedy and an extension to AFL, which can easily be applied to other existing CGFs. While this conclusion may not hold for programs that we did not use in the experiments, our evaluation shows the potential of remedial testing based on re-evaluation of covered paths.

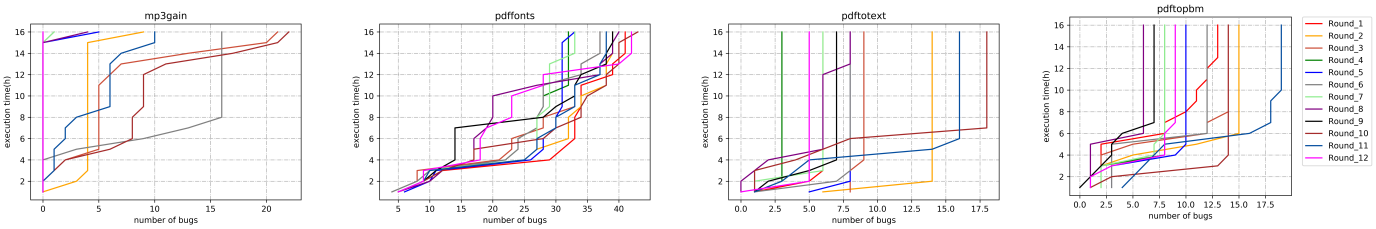

Figure 4. Number of bugs and execution time in exploration stage.

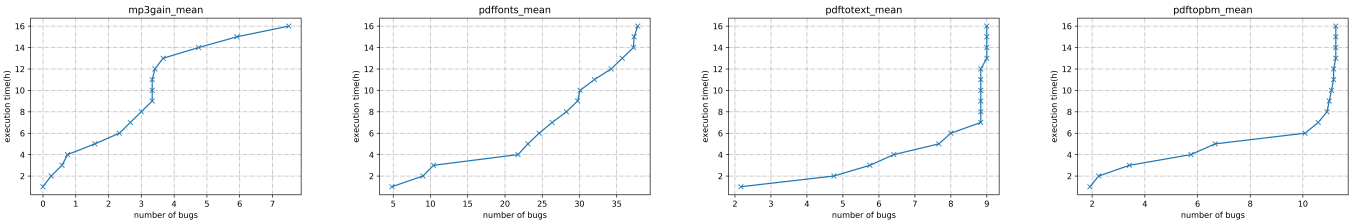

Figure 5. Average number of bugs and execution time in exploration stage.

6. Related Work

The mutation-based fuzzer uses actual inputs to continuously mutate the test cases in the corpus during the fuzzing process, and continuously feeds the target program. The code coverage is used as the key to measure the performance of the fuzzer. AFL [3] uses compile-time instrumentation and genetic algorithms to find interesting test cases, and can find new edge coverage based on these inputs. VUzzer [20] uses the "intelligent" mutation strategy based on data flow and control flow to generate high-quality inputs through the result feedback and by optimizing the input generation process. The experiments show that it can effectively speed up the mining efficiency and increase the depth of mining. FairFuzz [21] increases the coverage of AFL by identifying branches (rare branches) performed by a small amount of input generated by AFL and by using mutation mask creation algorithms to make mutations that tend to generate inputs that hit specific rare branches. AFLFast [12] proposes a strategy to make AFL geared toward the low-frequency path, providing more opportunities to the low-frequency path, which can effectively increase the coverage of AFL. LibFuzzer [4] uses SanitizerCoverage [22] to track basic block coverage information in order to generate more test cases that can cover new basic blocks. Sun et al. [23] proposed to use the ant colony algorithm to control seed inputs screening in greybox fuzzing. By estimating the transition rate between basic blocks, we can determine which the seed input is more likely to be mutated. PerfFuzz [24] generates inputs through feedback-

oriented mutation fuzzing generation, can find various inputs with different hot spots in the program, and escapes local maximums to have higher execution path length inputs. SPFuzzs [25] implement three mutation strategies, namely, head, content and sequence mutation strategies. They cover more paths by driving the fuzzing process, and provide a method of randomly assigning weights through messages and strategies. By continuously updating and improving the mutation strategy, the above research effectively improves the efficiency of fuzzing. As far as we know, in our experiment, if there are no new crashes for a long time (>*ct*), and it is undergoing the deterministic mutation operations at present, then it performs the next deterministic mutation or to enter the random mutation stage directly, which reduces unnecessary time consumption to a certain extent.

The generation-based fuzzer is significant for having a good understanding of the file format and interface specification of the target program. By establishing the model of the file format and interface specification, the fuzzer generates test cases according to the model. Dam et al. [26] established the Long Short-Term memory model based on deep learning, which automatically learns the semantic and grammatical features in the code, and proves that its predictive ability is better than the state-of-the-art vulnerability prediction models. Reddy et al. [27] proposed a reinforcement learning method to solve the diversification guidance problem, and used the most advanced testing tools to evaluate the ability of RLCheck. Godefroid et al. [28] proposed a machine learning technology based on neural networks to automatically generate grammatically test cases. AFL++ [29] provides a variety of novel functions that can extend the blurring process over multiple stages. With it, variants of specific targets can also be written by experienced security testers. Fioraldi et al. [30] proposed a new technique that can generate and mutate inputs automatically for the binary format of unknown basic blocks. This technique enables the input to meet the characteristics of certain formats during the initial analysis phase and enables deeper path access. You et al. [31] proposed a new fuzzy technology, which can generate effective seed inputs based on AFL to detect the validity of the input and record the input corresponding to this type of inspection. PMFuzz [32] automatically generates high-value test cases to detect crash consistency bugs in persistent memory (PM) programs. These efforts use syntax or semantic learning techniques to generate legitimate inputs. However, our work is not limited to using input format checking to screen legitimate inputs during the testing process, and we can obtain high coverage in a short time by using the corpus obtained by AFL test as the initial corpus in the exploration phase. Symbolic execution is an extremely effective software testing method that can generate inputs [33–35]. Symbolic execution can analyze the program to obtain input for the execution of a specific code area. In other words, when using symbolic execution to analyze a program, the program uses symbolic values as input instead of the specific values used in the general program execution. Symbolic execution is a heavyweight software testing method because the possible input of the analysis program needs to be able to obtain the support of the target source code. SAFL [36] is augmented with qualified seed generation and efficient coverage-directed mutation. Symbolic execution is used in a lightweight approach to generate qualified initial seeds. Valuable exploration directions are learned from the seeds to reach deep paths in program state space earlier and easier. However, for large software projects, it takes a lot of time to analyze the target source code. As ReFuzz is a lightweight extension of AFL, in order to be able to repeatedly reach the existing execution path, we choose to add the test that fails to generate a new path to the execution corpus to participate in subsequent mutations.

7. Conclusions

This paper designs and implements a remedy for saturation during greybox fuzzing, called ReFuzz. Using the corpus of the initial stage as the seed test inputs of the exploration stage, ReFuzz can explore the same set of execution paths extensively to find new and unique crashes along those paths within a limited time. The AFL directly feeds the input obtained by the mutation into the target program for running, which causes many non-

compliant seeds to be unable to explore deeper paths. In this paper, we proposed an input format checking algorithm that can filter the file conformed to the input format, which is beneficial to enhance the coverage depth of the execution path. At the same time, the mutation strategy we proposed can transition to the random mutation stage to continue testing when the deterministic mutation stage is stuck, which significantly accelerates the testing efficiency of fuzzing. We evaluated REFUZZ , using programs from prior related work. The experimental results show that REFUZZ can find new unique crashes that account for a large portion among the total unique crashes. Specifically, we discovered and submitted nine new vulnerabilities in the experimental subjects to the CVE database. We are in the process of analyzing and reporting more bugs to the developers.

In the future, in order to make our prototype tool better serviced in the real world, we will study how to combine machine learning to improve the efficiency of input format checking and design more complex automatic saturation strategies to strengthen the linkability of the tool. We will continue to improve REFUZZ to help increase the efficiency of fuzzers in the saturation state using parallel mode and deep reinforcement learning. We are planning to develop more corresponding interfaces and drivers to explore more vulnerabilities of IoT terminals for enhanced security of critical infrastructures.

Author Contributions: Conceptualization, D.Z. and H.Z.; methodology, D.Z.; software, Q.L.; validation, R.D.; writing—original draft preparation, Q.L. and H.Z.; writing—review and editing, Q.L.; supervision, D.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Fundamental Research Funds for the Central Universities through the project(No.2021QY010).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Frighetto, A. Coverage-Guided Binary Fuzzing with REVNG and LLVM Libfuzzer. Available online: https://www.politesi.polimi.it/bitstream/10589/173614/3/2021_04_Frighetto.pdf (accessed on 5 May 2021).
2. Sutton, M.; Greene, A.; Amini, P. *Fuzzing: Brute Force Vulnerability Discovery*; Addison-Wesley Professional: Boston, MA, USA, 2007.
3. American Fuzzy Lop. Available online: https://lcamtuf.coredump.cx/afl (accessed on 1 September 2020).
4. libFuzzer: A Library for Coverage-Guided Fuzz Testing. Available online: http://llvm.org/docs/LibFuzzer.html (accessed on 1 September 2020).
5. OSS-Fuzz: Continuous Fuzzing for Open Source Software. Available online: https://github.com/google/oss-fuzz (accessed on 1 October 2020).
6. Böhme, M.; Falk, B. Fuzzing: On the exponential cost of vulnerability discovery. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 8–13 November 2020 ; pp. 713–724.
7. Li, Y.; Ji, S.; Lv, C.; Chen, Y.; Chen, J.; Gu, Q.; Wu, C. V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing. CoRR. 2019. Available online: http://xxx.lanl.gov/abs/1901.01142 (accessed on 15 October 2020).
8. Godefroid, P. Fuzzing: Hack, art, and science. *Commun. ACM* **2020**, *63*, 70–76. [CrossRef]
9. Godefroid, P.; Klarlund, N.; Sen, K. DART: Directed automated random testing. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005; pp. 213–223.
10. Böhme, M.; Pham, V.T.; Nguyen, M.D.; Roychoudhury, A. Directed greybox fuzzing. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 30 October–3 November 2017; pp. 2329–2344.
11. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 711–725.
12. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* **2017**, *45*, 489–506. [CrossRef]
13. Yue, T.; Tang, Y.; Yu, B.; Wang, P.; Wang, E. Learnafl: Greybox fuzzing with knowledge enhancement. *IEEE Access* **2019**, *7*, 117029–117043. [CrossRef]
14. Pham, V.T.; Böhme, M.; Roychoudhury, A. AFLNet: A greybox fuzzer for network protocols. In Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, 24–28 October 2020; pp. 460–465.

15. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 15–19 October 2018; pp. 2095–2108.
16. Addresssanitizer. Available online: https://github.com/google/sanitizers/wiki/AddressSanitizer (accessed on 1 March 2021).
17. afl-Cmin. Available online: https://github.com/mirrorer/afl/blob/master/afl-cmin (accessed on 5 December 2020).
18. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Vigna, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In Proceedings of the Network and Distributed System Security Symposium, San Diego, California, USA, 21-24 February 2016.
19. Peng, H.; Shoshitaishvili, Y.; Payer, M. T-Fuzz: Fuzzing by Program Transformation. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 697–710. [CrossRef]
20. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. *NDSS* **2017**, *17*, 1–14.
21. Lemieux, C.; Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 475–485.
22. SanitizerCoverage. Available online: https://clang.llvm.org/docs/SanitizerCoverage.html (accessed on 2 February 2021).
23. Sun, B.; Wang, B.; Cui, B.; Fu, Y. Greybox Fuzzing Based on Ant Colony Algorithm. In Proceedings of the International Conference on Advanced Information Networking and Applications, Caserta, Italy, 15–17 April 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 1319–1329.
24. Lemieux, C.; Padhye, R.; Sen, K.; Song, D. Perffuzz: Automatically generating pathological inputs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, 16–21 June 2018; pp. 254–265.
25. Song, C.; Yu, B.; Zhou, X.; Yang, Q. SPFuzz: A Hierarchical Scheduling Framework for Stateful Network Protocol Fuzzing. *IEEE Access* **2019**, *7*, 18490–18499. [CrossRef]
26. Dam, H.K.; Tran, T.; Pham, T.T.M.; Ng, S.W.; Grundy, J.; Ghose, A. Automatic feature learning for predicting vulnerable software components. *IEEE Trans. Softw. Eng.* **2018**. [CrossRef]
27. Reddy, S.; Lemieux, C.; Padhye, R.; Sen, K. Quickly generating diverse valid test inputs with reinforcement learning. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), Seoul, Korea, 5–11 October 2020; pp. 1410–1421.
28. Godefroid, P.; Peleg, H.; Singh, R. Learn&fuzz: Machine learning for input fuzzing. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana-Champaign, IL, USA, 30 October–3 November 2017; pp. 50–59.
29. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. AFL++: Combining incremental steps of fuzzing research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT), Online, 11 August 2020.
30. Fioraldi, A.; D'Elia, D.C.; Coppa, E. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, 18–22 July 2020; pp. 1–13.
31. You, W.; Liu, X.; Ma, S.; Perry, D.; Zhang, X.; Liang, B. SLF: fuzzing without valid seed inputs. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 712–723.
32. Liu, S.; Mahar, S.; Ray, B.; Khan, S. PMFuzz: Test case generation for persistent memory programs. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 19–23 April 2021; pp. 487–502.
33. Cadar, C.; Dunbar, D.; Engler, D.R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the OSDI, Berkeley, CA, USA, 8–10 December 2008; Volume 8, pp. 209–224.
34. Ge, X.; Taneja, K.; Xie, T.; Tillmann, N. DyTa: Dynamic symbolic execution guided with static Verification results. In Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA, 21–28 May 2011; pp. 992–994.
35. Chen, J.; Hu, W.; Zhang, L.; Hao, D.; Khurshid, S. Learning to Accelerate Symbolic Execution via Code Transformation. In Proceedings of the ECOOP, Amsterdam, The Netherlands, 16–21 July 2018.
36. Wang, M.; Liang, J.; Chen, Y.; Jiang, Y.; Jiao, X.; Liu, H.; Zhao, X.; Sun, J. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Gothenburg, Sweden, 27 May–3 June 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 61–64. [CrossRef]