Improving Seed Quality with Historical Fuzzing Results

Yang Li[a], Yingpei Zeng[a,*], Xiangpu Song[b] and Shanqing Guo[b,*]

[a]*School of Cyberspace, Hangzhou Dianzi University, Hangzhou 310000, China*
[b]*School of Cyber Science and Technology, Shangdong University, Qingdao 266237, China*

ARTICLE INFO

Keywords:
Fuzzing
History information
Corpora minimization
Software testing

ABSTRACT

Context: Coverage-guided fuzzing (CGF) has achieved great success in discovering software vulnerabilities. The efficiency of CGF highly relies on the quality of the initial seed corpus. Although there have been some works in recent years investigating the initial seed selection, usually only the corpus given by developers or downloaded from the Internet is used to get the initial seed corpus.
Objective: We assess several existing corpus minimization tools and find that none of them effectively leverage information contained in historical fuzzing results. The historical fuzzing results may come from previous fuzz testing or the emerging continuous fuzzing integration in the software development cycle. Therefore, we want to utilize history fuzzing results to generate a high-quality initial corpus to enhance the fuzzing performance. Besides, the size of the initial corpus will affect the fuzzing efficiency, so using a minimization tool to extract valuable seeds from historical results is essential.
Method: We propose to use historical fuzzing results to help construct the initial seed corpus and further develop a corpus minimization tool named MCM (multiple corpora minimization), which can analyze multiple fuzzing results and use information including edge appearance frequency to help seed selection.
Results: We implement a prototype of MCM and evaluate it on 10 open-source programs. Our experiments show that by using historical fuzzing results to expand the size of the initial seed corpus even a small number, e.g., from 20 to only 100, the branch coverage improves up to 14%. Meanwhile, MCM can achieve higher code coverage than existing corpus minimization tools, including AFL-CMIN and OPTIMIN.
Conclusion: Our study shows using historical results to generate a high-quality initial corpus is practical and can effectively improve the fuzzing performance.

1. Introduction

Coverage-guided fuzzing (CGF) has become one of the most prevalent software-testing methods in recent years [8, 34, 50, 68, 65]. Since vulnerabilities are typically activated through the execution of specific code paths, CGF aims to explore more distinct code paths to identify novel vulnerabilities. Importantly, the coverage information is dynamically updated when a new code path is executed, and such information could guide the CGF fuzzer to explore the uncovered part of the program under test. The initial seed corpus is known to be important to the success of a fuzzing campaign [23], as the developers of the Mozilla Firefox browser remark: *"Success of fuzzing really stands and falls with the quality of the samples. If the originals don't cover certain parts of the implementation, then the fuzzer will also have to do more work to get there"* [38]. Users usually obtain initial seeds from the program developers (e.g., finding seeds in code repositories) or the Internet (e.g., searching for files in given formats), and then minimize the corpus with a corpus minimization tool like AFL-CMIN [28, 35, 23].

[★]This work was supported by Zhejiang Provincial Natural Science Foundation of China under Grant No. LY22F020022, the National Natural Science Foundation of China under Grant No. 61902098, the National Natural Science Foundation of China under Grant No. 62372268, Shandong Provincial Natural Science Foundation under Grant No. ZR2021LZH007 and No. ZR2022LZH013, Jinan City "20 New Universities" Funding Project under Grant No. 2021GXRC084

[*]Corresponding author

`chrisly@hdu.edu.cn` (Y. Li); `yzeng@hdu.edu.cn` (Y. Zeng); `songxiangpu@sdu.edu.cn` (X. Song); `guoshanqing@sdu.edu.cn` (S. Guo)

Since a program may be fuzzed multiple times now, it is possible to generate a high-quality corpus from the historical fuzzing results. For example, users may fuzz different versions of a program to check whether there are vulnerabilities in them. In OSS-Fuzz [49], many open-source programs are being fuzzed day after day to detect bugs timely. Also, fuzz testing is being added as a phase in continuous integration (CI) [14, 2] in the software development cycle. All of the fuzz testing could produce multiple historical fuzzing results for a program. If we could extract useful seeds from them, like successfully leveraging historical information in fields like transfer learning [59] or reinforcement learning [22], we could explore the program more efficiently in later fuzz testing. Hence, we propose to use historical fuzzing results to enhance the quality of the initial corpus. Given the large number of seeds from historical fuzzing results, it is necessary to employ a minimization tool to reduce or select a subset of the historical fuzzing results to get useful seeds.

However, corpus minimization tools do not leverage all important information contained in historical fuzzing results. Recent research [62] has shown that the size of the initial corpus will affect the efficiency of fuzzing. Thus, it is important to keep the initial corpus reasonably small. Existing tools commonly use edge tuple information (also known as edge information), which is defined by AFL [64], as the basis for minimization. Each program can be represented as a control flow graph, where each node corresponds to a basic block, and edges represent the jumps between these blocks. One popular tool, AFL-CMIN [64], employs a greedy minimization algorithm to select seeds containing rare edge tuples. Another tool, OPTIMIN [19], formulates corpus minimization as a Maximum Satisfiability Problem (MAX-SAT) [26] to select seeds. Both fail to consider the edge relationships across different fuzzing campaigns (or say runs). Intuitively, an edge that occurs in each historical fuzzing campaign is more likely to be discovered compared to that only appears in a single fuzzing campaign, which means the edges that occur in a single fuzzing campaign should be retained first during the seed selection. Other information like the time an edge first appeared in fuzzing should also be considered, since the later discovered edges may be harder to find in fuzzing.

To better utilize the valuable information in historical fuzzing results, we developed a new minimization tool, MCM, to extract more efficient seeds from these results. First, MCM analyzes the coverage information and computes edge frequency across multiple fuzzing results to obtain the edge differential information. Next, it uses two different algorithms to select seeds from the historical fuzzing results. The two algorithms are proposed based on the similar ideas of two prevalent minimization tools AFL-CMIN and OPTIMIN but modified to better select seeds from historical fuzzing results. For MCM-CMIN, we choose seeds with rare and later-appearance edges and add corresponding seeds with faster execution speed into the corpus first, while MCM-OPT tries to preserve more rare edges using the MAX-SAT solver. By utilizing edge differential information across different fuzzing campaigns, along with seed execution speed and edge first appearance time, MCM can successfully generate a compact corpus containing more valuable seeds.

We compare MCM with AFL-CMIN and OPTIMIN on 10 open-source Linux programs. Our experiments show that by reusing the historical results from the same program to enlarge the initial corpus from 20 to only 100, the branch coverage improves up to 14%. Besides, MCM-CMIN outperforms MCM-OPT, AFL-CMIN, and OPTIMIN on most of the programs when reusing historical fuzzing results. Meanwhile, we chose four different commits and versions to verify the effectiveness of using historical results on slightly changed programs, and results show that the branch coverage improves up to 14.9%.

In summary, we make the following main contributions:

- We propose to leverage historical fuzzing results to help obtain initial seeds and demonstrate the method can improve fuzzing efficiency on real programs.
- We propose a new seed corpus minimization tool called MCM, which selects seeds from historical fuzzing results based on the frequency and appearance time of edges etc.
- We compare MCM with state-of-the-art corpus minimization tools AFL-CMIN and OPTIMIIN on 10 open-source programs. The results demonstrate that MCM outperforms other corpus minimization techniques. We will publish our code at `https://github.com/loren998/MCM`.

The remainder of this paper is structured as follows. Section 2 introduces the background and limitations of the field. Section 3 discusses the reusing of historical results and existing problems. Section 4 and Section 5 present the design and implementation details of MCM. Results and analysis of the study are covered in Section 6. Section 7 discusses some limitations and observations of our work, and Section 8 presents the related work before we conclude our paper in Section 9.

2. Background and limitations

2.1. Coverage-guided Fuzzing

Coverage-guided Fuzzing (CGF) [64, 13, 28] has proven to be highly effective in locating bugs and vulnerabilities in real-world software. The primary objective of CGF is to explore as many code paths as possible to identify potential weaknesses. Fuzzers employ various generation and mutation strategies to generate random and even malicious inputs for testing the target program (i.e., the program under test, PUT). Due to the challenges associated with directly observing the internal state of the target program, coverage feedback is used to gauge the novelty of an execution. After a significant duration of fuzzing runtime, the corpus will comprise numerous interesting seeds, referred to as historical fuzzing results after fuzz testing.

Figure 1 illustrates the fundamental workflow and key stages of CGF. These stages can be divided into two parts: pre-processing and scheduling. The pre-processing stage manages the corpus set before or during the fuzzing process, while the scheduling is responsible for the primary fuzzing execution procedure.

Pre-processing. As a well-structured seed set significantly improves the fuzzing efficiency, existing works employ various methods to enhance the quality of the initial corpus. Part of the existing fuzzers [32, 54, 15] utilizes the corpus minimization tool AFL-CMIN to process the corpus set, which is either by crawling the internet or provided by developers of the target program. Another choice is using the benchmark platforms [36] which provides seeds for the testing. Nyx-net [48], StateAFL [39] and NSFuzz [45] test their fuzzers on Profuzzbench [40] and EMS [33], FuzzJIT [55] and Rapidfuzz [63] deploy their experiments on the UNIFUZZ [28]. The efficiency of fuzzing is significantly influenced by the quality of the initial corpus, as highlighted in previous studies [23]. Therefore, the pre-processing stage assumes great importance within the overall fuzzing process.

Scheduling. Scheduling is another crucial stage of the fuzz testing responsible for managing the entire fuzzing process. Throughout the fuzzing procedure, the fuzzer randomly selects or prioritizes the most intriguing seed from the seed queue, applying one or multiple mutations to generate new inputs. Subsequently, the fuzzer tests the target program with the newly generated input and continuously observes the coverage feedback. Concurrently, the monitor assesses code coverage changes to determine if a new path is executed or a new crash is detected. Any input resulting in code coverage

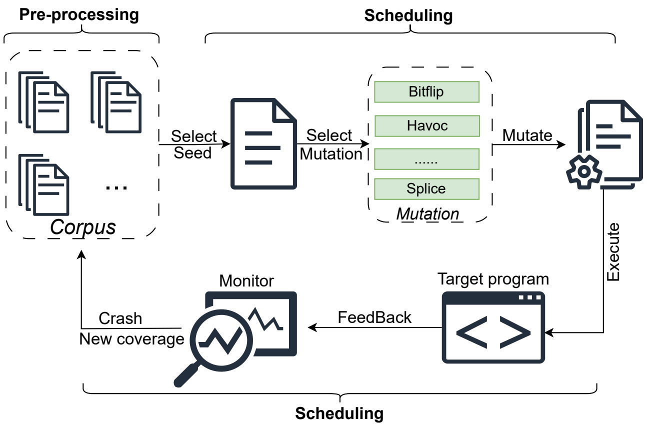


Fig. 1: Overview of Coverage-guided fuzzing.

alteration is considered interesting and will be added to the seed corpus for further fuzzing.

2.2. AFL-CMIN and OPTIMIN

Due to the popularity of AFL [64], AFL-CMIN is one of the most commonly used corpus minimization tools. It utilizes a heuristic algorithm to create a smaller corpus that covers all the edges. Specifically, AFL-CMIN prioritizes edge tuples with rare occurrences, selects the smallest input file for each edge tuple, and stores the corresponding seed. By employing a greedy minimization algorithm and considering both file size and edge frequency, AFL-CMIN proves highly effective in corpus minimization.

In contrast to AFL-CMIN, which employs a heuristic algorithm, OPTIMIN [19] encodes the corpus minimization as a maximum satisfiability problem. OPTIMIN treats edges as hard constraints, ensuring that the solver must include every edge in the solution. Additionally, it treats the exclusion of specific seeds as a soft constraint, allowing the solution to select a minimal number of seeds. To achieve this, OPTIMIN employs EvalMaxSAT solver [4] to get the solution. By generating the solution with the fewest number of seeds, OPTIMIN proves itself to be an effective corpus minimization tool.

2.3. Limitations of existing approaches

We analyze four programs of five historical fuzzing results, as is shown in Table 1. Since the randomness of fuzz testing, new edges appear each time different control flows are explored. We can see that different fuzzing campaigns commonly contain some unique edges which means that they trigger different control flows. If rare edges have been identified in previous fuzzing results, adding the corresponding seed to the initial corpus can help trigger unique paths or crashes. Traditional minimization tools retain all edges while reducing their number to improve corpus quality. However, for programs where most of the edges need to be triggered by different seeds, the resultant solution is still excessively large. Xu et al. [62] demonstrated that time spent on opening/closing test cases introduces a 2× overhead, which decreases the efficiency of fuzz testing. Additionally, treating multiple fuzzing results as a whole will lose some crucial information across different fuzzing campaigns (e.g., edge appearing frequency across different fuzzing campaigns). Hence, using corpus minimization tools to select part of rare edges instead of holding all the edges is a better substitute.

Table 1
Unique edges in different fuzzing campaigns.

Target	R1	R2	R3	R4	R5
pdfimages	55	106	31	182	13
tcpdump	103	451	208	319	383
nm	43	76	133	70	19
objdump	104	33	18	35	23

Let's illustrate with the following example: We have 10 fuzzing results where edge $E1$ appears in only one campaign 15 times and edge $E2$ appears once in every campaign. When AFL-CMIN treats the 10 fuzzing results as a whole, it prioritizes edge $E2$ over edge $E1$ due to its rarity. While OPTIMIN treats $E1$ and $E2$ equally, which loses the edge's priority information. However, we can assert that if an edge appears in every fuzzing result, it is more likely to continue appearing in subsequent campaigns, as it is likely part of the main control flow. Even though E1 has a higher occurrence frequency (e.g., the corresponding seed has more descendant seeds), if it only appears in a single campaign, it is more likely part of a rare control flow and is difficult to trigger. However, both AFL-CMIN and OPTIMIN fail to prioritize the rarest edges and corresponding seeds, which decreases the quality of the initial corpus.

3. History reusing for corpus generation

In this section, we introduce the structure of using minimization tools to generate a high-quality initial corpus using historical fuzzing results and some limitations of using existing minimization tools.

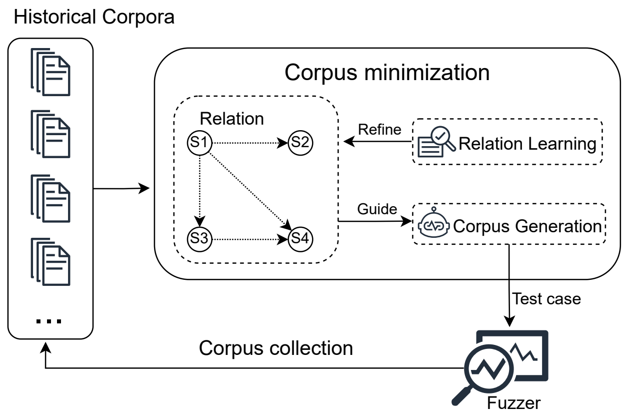


Fig. 2: The workflow of initial corpus generation using historical fuzzing corpora.

Figure 2 shows the overall workflow of historical fuzzing results (historical corpora) reusing to generate a high-quality initial corpus. To effectively minimize the multiple corpora, the relation learning process refines the concept of edge frequency by computing edge appearances across different fuzzing campaigns, and prioritizing the identification of rare control flows. The newly generated edge frequency will then be used to aid in seed selection, which will subsequently generate the new corpus. The corpus is then employed for fuzzing over a user-specified duration, and the final corpus will be collected for future historical corpora reusing.

Both AFL-CMIN and OPTIMIN can only minimize a single corpus, so historical corpora must be merged into one integrated corpus. AFL-CMIN's relation learning is based on edge frequency, while OPTIMIN defaults to whether an edge is executed.

By using a heuristic algorithm, AFL-CMIN successfully selects seeds one by one until all edges are covered. However, due to the excessive number of seeds in multiple corpora, OPTIMIN often fails to generate a minimized solution because of the limitations of the SAT solver. One solution is to use OPTIMIN to minimize each corpus individually before merging them into an integrated corpus, and then use OPTIMIN again to generate the final minimized corpus. Another approach is to minimize one corpus with OPTIMIN, merge it with the next corpus, and repeat this process until all corpora are merged. Both solutions may result in a non-globally optimal minimized corpus, as some interesting seeds might be filtered out during the simplification process.

While the minimized solution can be generated from the consolidated corpus, the differential information across multiple fuzz testing campaigns is disregarded, as the example mentioned above in Section 2.3, which will decrease the quality of the initial corpus. Besides the initial corpus generated by AFL-CMIN and OPTIMIN can still be too large which is not practical to use. Therefore, we believe that leveraging differential information from various fuzzing campaigns can aid in relation learning to select more valuable seeds from historical corpora. To better reuse historical corpora, we modified AFL-CMIN and OPTIMIN and proposed our MCM to select more effective seeds.

4. Design of MCM

This section elaborates on the keys of MCM and details two new minimization algorithms MCM-CMIN and MCM-OPT modified based on AFL-CMIN and OPTIMIN. Let τ be one fuzzing corpus, while κ represents historical corpora which consist of a set of τ.

4.1. Keys of MCM

One of the keys to MCM is differential information across multiple historical corpora. To be specific, the frequency at which an edge appears among different historical fuzzing results is important since we can find in Table 1 that different fuzzing results usually contain some unique edges. To avoid ambiguity, we refer to it as the edge appearance frequency in the rest of the article. Historical corpora typically contain various edges, but not all edges are equally important, and we should focus more on the rare ones. Existing corpus minimization tools consider the edge frequency instead of the edge appearance frequency across different fuzzing campaigns. As the example in section 2.3 shows, this will undermine the quality of the minimized corpus if we restrict the size of the initial corpus. Consequently, it becomes necessary to calculate the edge appearance frequency across different fuzzing campaigns to enhance the seed selection.

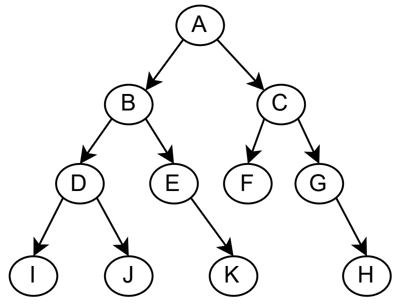
Edge debut time is also an important factor, it can indicate the difficulty an edge is to explore because rare edges usually appear later. Edge debut time is the time corresponding to the earliest appearance of the seed containing this edge. It can be represented by the relative time from the beginning of the fuzz testing to the seed that first contains this edge in the historical fuzzing results. When a new edge is detected, the fuzzer will preserve the corresponding seed to the corpus immediately. The successor of this seed will likely contain the same newfound edge, which means only the earliest time the edge appears is of importance. Therefore, we can prioritize edges with longer time intervals and save corresponding seeds to make the minimization more effective.

Another crucial aspect to consider is the seed execution speed. The more inputs the target program executes, the more likely the fuzzer will uncover new paths. Slow execution speeds of chosen seeds can hinder the number of inputs tested by the target program. Conversely, opting for seeds with higher execution speeds can facilitate the testing of a larger number of inputs. We incorporate seed execution speed instead of file size since we think seed execution speed is a more accurate indicator.

Algorithm 1: MCM Relation Learning

```
   Input: Historical corpora $\kappa$
   Output: EdgeCount, EdgeMap
1  for $\tau \in \kappa$ do
2      for Seed $\in \tau$ do
3          SeedCov, SeedSpeed ← ShowMap(Seed)
4          for Edge $\in$ SeedCov do
5              SeedInfo ← Pair(Seed.SeedID, SeedSpeed)
6              EdgeMap[Edge.EdgeID].add(SeedInfo)
7              EdgeSet.add(Edge.EdgeID)
8          end
9      end
10     for Edge $\in$ EdgeSet do
11         EdgeCount[Edge.EdgeID].add(1)
12     end
13 end
```

MCM's relation learning is shown in Algorithm 1. AFL-CMIN takes edge frequency into account as its relation learning and OPTIMIN uses whether an edge is executed as its relation learning. Different from the above two relation learning, MCM learns edge appearance frequency across multiple historical corpora to get differential information. To be specific, for every corpus τ, MCM learns similar information as OPTIMIN does. MCM uses SeedInfo consisting of SeedID and SeedSpeed for getting edge debut time and seed execution speed (line 5). When an edge appears earlier,



(a) Program control-flow graph

Corpus 1:
S1 A→B→E→K
S2 A→C→F
A→C→G→H

Corpus 2:
S3 A→C→F
A→B→D
S4 A→B→D→I

Corpus 3:
S5 A→B
A→C→G→H
S6 A→B→D→J

***(b) Seed traces.* The corpus *c* is considered as simple historical results, each seed *s* within the corpus represents a seed, and the sequence following the seed is viewed as a simplified control flow.**

MCM-OPT constraints

Edges:

(A,B):	S1 ∨ S3 ∨ S4 ∨ S5 ∨ S6	We_{min}
(A,C):	S2 ∨ S3 ∨ S5	We_{min}
(B,D):	S3 ∨ S4 ∨ S6	We_{min}
(C,F):	S2 ∨ S3	We_{min}
(C,G), (G,H):	S2 ∨ S5	We_{min}
(B,E), (E,K):	S1	We_{max}
(D,I):	S4	We_{max}
(D,J):	S6	We_{max}

Seeds

¬ S1, ¬ S2, ¬ S3, ¬ S4, ¬ S5, ¬ S6 Ws

***(c) MCM-OPT CNF constraints.* MCM-OPT treat both edges and seeds constraints as soft constraints to select the most important seeds from the corpus. Each edge is encodeed as a disjunction of seeds that cover that edge, and will assign weight according to appearance frequency across different corpus, ranging from We_{min} to $We_{max.}$ Each seed is assigned a negation clause with weight Ws to minimize the corpus size.**

Fig. 3: MCM-OPT historical corpora reusing.

its corresponding number in SeedID will be smaller. MCM uses the number, which is the serial number in the corpus generated by the fuzzer when the seed is added to the corpus during the fuzzy testing process, to get the edge debut time. We further calculate the appearance frequency of an edge appearing in different historical corpora (lines 10-12), and then we can get the edge appearance frequency. So, for both MCM-OPT and MCM-CMIN, their first step will be MCM relation learning to get the edge appearance frequency and other key information mentioned above.

4.2. MCM-OPT

Similar to OPTIMIN, MCM-OPT also encodes the seed selection to a MAX-SAT problem. Since multiple fuzzing results consisted of excessive edges, it is unfeasible to save all the edges in the solution, which otherwise would lead to the initial corpus of impractical size and relatively low quality. Hence, we modify OPTIMIN and propose the design of MCM-OPT, as is shown in Algorithm 2 to consider edges whose appearance frequency is smaller than half of the corpora number as important instead of maintaining all the edges.

After MCM's relation learning, MCM-OPT generates weighted MAX-SAT soft constraints for each edge and seed (lines 2-13). For all seeds that contain a specific edge, MCM-OPT uses the OR operation to combine them to form an edge expression (line 6), so that at least one of these seeds is retained. Besides, it sets a negation clause for each seed with weight Ws to minimize the corpus size (line 7).

Algorithm 2: MCM-OPT Seed Processing

```
   Input: Historical corpora κ, EdgeCount, EdgeMap
   Output: SeedCorpus (minimized seed corpus)
1  // Step1: Generate weighted MAX-SAT soft constraints
2  for [EdgeID, SeedsInfo] ∈ EdgeMap do
3      EdgesExpr ← FALSE
4      for SeedInfo ∈ SeedsInfo do
5          CurSeedID ← SeedInfo.SeedID
6          EdgesExpr.OR(CurSeedID)
7          SeedClause ← GENSEEDCLAUSE(!CurSeedID, SeedWeight)
8          Solver.ADD(SeedClause)
9      end
10     EdgeWeight ← ASSIGNWEIGHT(EdgeCount[EdgeID])
11     EdgeClause ← GENEDGECLAUSE(EdgesExpr, EdgeWeight)
12     Solver.ADD(EdgeClause)
13 end
14 // Step2: Solve weighted MAX-SAT problem and generate corpus
15 SolverResult ← CHECK(Solver)
16 if ISSAT(SolverResult) then
17     for SeedClause ∈ AllClauses do
18         if SeedClause.ISTRUE then
19             Seed ← GETSEED(SeedClause.SeedID, κ)
20             SeedCorpus.ADD(Seed)
21         end
22     end
23 end
24 else
25     SeedCorpus ← κ
26 end
```

MCM-OPT then assigns the weight for every edge according to its edge appearance frequency across different fuzzing campaigns (lines 10-11), ranging from $We_{\min}$ to $We_{\max}$. The weight assignment is performed to tell the solver the loss of disregard specific edge when addressing the satisfiability problem. For edges that only appear in a single campaign, a weight of $We_{\max}$ is assigned, indicating that discarding this edge would result in the maximum loss. Conversely, if an edge appears more frequently across multiple campaigns, a lower weight is assigned. Currently, we only use $We_{\min}$ and $We_{\max}$ to make the satisfiability problem easy to solve.

Finally, MCM-OPT employs the SAT solver to solve the weighted MAX-SAT problem (lines 15-26). The solver evaluates whether a solution exists. If the solver's state is SAT, indicating a minimized corpus can be generated, it incorporates seeds into the corpus in which the corresponding clauses are satisfied (lines 18-21). Otherwise, the SAT solver can not generate a solution, and all the seeds will be preserved. Since MCM-OPT treats the chosen seeds equally, it will select seeds randomly to meet the size requirement if the user predefined the size threshold of the initial corpus.

For illustrative purposes, consider the following example in Figure 3 involving three historical corpora reusing. As the corpora number of this example is three, that means MCM-OPT only considers those edges whose appearance frequency is one as important. So edges $(B, E), (E, K), (D, J)$, and (D, I) are treated as important and will be assigned with $We_{\max}$. While other edges are relatively unimportant and will be assigned with $We_{\min}$. So, some example edge clauses can be expressed as Equation 1 and Equation 2. For each seed, $S_1 - S_6$, MCM-OPT generates seed clauses like Equation 3.

$$Clause\_(B, E) = (S1, We_{\max}) \tag{1}$$

$$Clause\_(A, C) = (S2 \vee S3 \vee S5, We_{\min}) \tag{2}$$

$$Clause\_(S_1) = (!S1, Ws) \tag{3}$$

After generating the edge and seed clauses, MCM-OPT can then use SAT-Prover to generate the solution. since MCM-OPT only considers $(B, E), (E, K), (D, J)$, and (D, I) as important, then the generated corpus will be S_1, S_4, S_6. Since OPTIMIN needs to maintain all the edges, its solution will be S_1, S_4, S_6 and S_2. However, we can find edges in S_2 appear in more than half of the corpora number, which is easily to be explored.

Nevertheless, due to the limitation of the SAT solver, encoding factors such as the edge debut time and the seed execution speed into seed selection will result in the solver failing to generate a solution. Thus, we introduce MCM-CMIN to optimize the utilization of MCM's important factors.

4.3. MCM-CMIN

MCM-CMIN employs a heuristic greedy approach for seed selection, as is shown in Algorithm 3. Similar to AFL-CMIN, MCM-CMIN also tries to find a locally optimal solution. But MCM-CMIN replaces the edge frequency with edge appearance frequency across different fuzzing campaigns and uses other factors like edge debut time to help seed selection. MCM-CMIN implements a greedy algorithm to select the best seeds one by one based on the MCM's key factors until the required number of the initial corpus is achieved.

After getting MCM's relation learning information, MCM-CMIN employs edge appearance frequency and edge debut time to identify the best candidate edge (lines 5-17). MCM-CMIN will first generate a container for storing edge and seed information from the EdgeMap and EdgeCount (lines 5-15). To be specific, it will get the edge appearance frequency from EdgeCount (line 5). Meanwhile, MCM-CMIN will compare the SeedInfo from the EdgeMap and use the corresponding number in SeedID as the indicator of the relative time interval to get the edge debut time (lines 8-12). Then the container will be sorted with edge appearance frequency in ascending order first. If edges with the same appearance frequency, MCM-CMIN sorts with edge debut time in descending order (line 16). After sorting, the first edge in the container is the optimal edge (line 17).

Upon selecting the optimal edge, the candidate seeds can be further filtered based on the execution speed to choose the seed with the highest execution speed (lines 20-25). Finally, MCM-CMIN incorporates the chosen seed into the corpus and updates the edge map and edge count to eliminate the edges that have been covered by the selected seed (lines 29-33). This process is reiterated until the edge map is empty or the size of the seed corpus meets the user-predefined threshold.

Consider the following example, there are five historical fuzzing results and three edges, $E1$, $E2$, and $E3$. Edge $E1$ appears in only one campaign and in seeds $S1, S2, S7, S9$, and edge $E2$ appears in three campaigns and is contained in seeds $S2, S4, S6, S8, S10$, while $E3$ appears in one campaign and in seeds $S5, S9, S10$. Therefore, MCM-CMIN will calculate the appearance frequency of $E1, E2, E3$ as 1, 3 and 1, respectively. MCM-CMIN prioritizes $E1$ and $E3$ due to their lower appearance frequency. Then, it iterates through the corresponding candidate seeds $S1, S2, S7, S9$ of $E1$ and $S5, S9, S10$ of $E3$ to choose the optimal edge according to their debut time. Assuming that the debut time of $E1$ is earlier, MCM-CMIN will choose the best seed from seeds $S5, S9, S10$. Suppose that $S5$ has the fastest execution speed, then $S5$ will be incorporated into the corpus and MCM-CMIN will remove the edges contained in $S5$ from the edge map and edge count. Subsequently, it will select $E1$ to repeat the above process and finally $E2$. AFL-CMIN's edge selection order will be $E2$, $E1$, and $E3$ due to their edge frequency, decreasing the corpus quality when the corpus size is limited.

Algorithm 3: MCM-CMIN Seed Processing

```
Input: Historical corpora $\kappa$, EdgeCount, EdgeMap, UserFixedNumber
Output: SeedCorpus (minimized seed corpus)
for EdgeMap $\neq \emptyset$ do
    EdgeInfoVec ← NULL
    FastSpeed ← INT_MAX, FastSeed ← NULL
    // Step1: choose the optimal edge based on edge appearance
      frequency and debut time
    for [EdgeID,EdgeFreq]∈ EdgeCount do
        SeedsInfo ← EdgeMap[EdgeID]
        EdgeDebutTime ← INT_MAX
        for SeedInfo ∈ SeedsInfo do
            if RelativeID(SeedInfo.SeedID) < EdgeDebutTime
              then
                EdgeDebutTime ← RelativeID(SeedInfo.SeedID)
            end
        end
        PairInfo ← Pair(EdgeID, (EdgeDebutTime, EdgeFreq))
        EdgeInfoVec.push(PairInfo)
    end
    EdgeInfoVec.sort(EdgeFreq, EdgeDebutTime)
    InterstingEdgeID ← EdgeInfoVec[0].EdgeID
    // Step2: Select seed based on execution speed
    CandidateSeeds ← EdgeMap[InterstingEdgeID]
    for SeedInfo ∈ CandidateSeeds do
        if SeedInfo.SeedSpeed < FastSpeed then
            FastSpeed ← SeedInfo.SeedSpeed
            FastSeed ← SeedInfo
        end
    end
    Seed ← GetSeed(FastSeed.SeedID)
    SeedCorpus.add(FastSeed)
    // Step3: Update the EdgeMap and EdgeCount
    RemoveEdges ← ShowMap(FastSeed)
    for SeedEdge ∈ RemoveEdges do
        EdgeMap.remove(SeedEdge.EdgeID)
        EdgeCount.remove(SeedEdge.EdgeID)
    end
    if SeedCorpus.size() = UserFixedNumber then
        Break
    end
end
```

5. Implementation of corpus minimization

We implement a prototype of MCM-CMIN and MCM-OPT based on AFLplusplus-4.00c [13] and reuse some code of OPTIMIN [19] to solve the seed selection problem. The implementation has approximately 1,700 lines of C++ code and we choose Z3Prover [11] to solve the SAT problem.

Corpus size. The size of the initial corpus is not a fixed value and can be specified by the user. To establish a suitable default corpus size, We conducted a survey and analysis of 15 academic fuzzers, as shown in table 2. Although some fuzzers do not specify the size of the initial corpus, most others typically set the initial corpus size to 100. Meanwhile, in a widely used fuzzer benchmarking framework Fuzzbench [36], 85% of its benchmark programs use the corpus with a size smaller than 100. Furthermore, among the 135 projects that provided initial corpus for fuzzing in Google's OSS-Fuzz, half of them also consist of less than 100 seeds. Consequently, our decision to adopt a default corpus size of 100 for different corpus minimization tools is practical.

Clause weight. In MCM-OPT, we generate the seeds clause and edges clause to solve the satiftifity problem. For every seed, we use the default soft clause weight of one, which means the solver gets an instruction that if the solution does not include this seed, it will make a loss of one. For the edges clause, edge appearance frequency smaller than half of the fuzzing campaigns is regarded as important, and other edges are relevantly unimportant. To those important edges, we set the weight of 100 to each edge clause, and the weight of the other edge clause is the same as the seed clause.

Table 2
Initial corpus size of academic fuzzers. We adopt the categories and notation used by Klees et al. [23] and Herrera et al. [19]: R means randomly sampled seeds; M means manually constructed seeds; G means automatically generated seed; V means the paper assumes the existence of valid seed(s), but with unknown provenance; V* means valid seed(s) with known provenance.

Paper	Seed	Size
SNPSfuzzer [27]	V	–
DARWIN [21]	V*	–
EMS [33]	R/V*	100
MOPT [32]	R	100
Nezha [44]	R	100
Coverset [47]	R	100
CollAFL [16]	V*	–
T-fuzz [43]	V*	–
Greyone [15]	R	10
Tortoisefuzz [58]	R	1
FUZZSIM [61]	R/V	100
SYMFUZZ [7]	M	1
SeededFuzz [57]	R/G	20
Skyfire [53]	R/G	650
UniFuzz [28]	V*	100

6. Evaluation

We evaluate our proposed design by answering the following questions:

- **RQ1.** What is the impact of using historical fuzzing results? (§6.3, §6.4)
- **RQ2.** Why does the performance of MCM differ from other minimization tools? (§6.5)
- **RQ3.** What is the impact of using historical fuzzing results to fuzz slightly changed programs? (§6.6)

6.1. Experiment Design

In this section, we first assess the effectiveness of MCM in leveraging multiple historical fuzzing results to generate the initial corpus. MCM should be able to cover discovered execution paths and vulnerabilities in past historical fuzzing campaigns as much as possible while triggering more new paths and vulnerabilities. To evaluate this, we designed two types of experiments (§6.3, §6.4), the first experiment compares the fuzzing results using the corpus generated from historical data against the original version, aiming to verify that using historical fuzzing results to generate the initial corpus can improve fuzzing efficiency. The second experiment compares code coverage and vulnerability discovery capabilities with AFL-CMIN and OPTIMIN, demonstrating that MCM can effectively learn critical information across different fuzzing campaigns and generate a high-quality corpus.

Table 3
The configuration of test programs.

Target	Version	Invocation	Format
openssl	openssl-1.1.1i	./server	packet
readpng	libpng-1.6.37	./readpng	png
cxxfilt	binutils-2.35.1	./cxxfilt	text
pdfimages	xpdf-4.02	./pdfimages @@ /dev/null	pdf
libxml	libxml2-v2.9.10	./xmllint @@	xml
readelf	binutils-2.35.1	./readelf -a @@	elf
infotocap	ncurses-6.2	./infotocap @@	text
tcpdump	tcpdump-4.9.3	./tcpdump -ee -vv -nnr @@	pcap
nm	binutils-2.35.1	./nm-new -A -a -l -S -s -C @@	elf
objdump	binutils-2.35.1	./objdump -xsSD @@	elf

In addition to the above experiments, we also compare and analyze the initial corpus generated by MCM, AFL-CMIN, and OPTIMIN against the original historical results (§6.5). Specifically, We examine the size of the minimized corpus produced by different tools to assess the compression efficiency of different minimization methods. Furthermore, We analyze the edge distribution within the generated corpora when the corpus size is limited, aiming to understand how edge appearance frequency impacts the quality and effectiveness of the corpus.

The last experiment (§6.6) aims to verify the effectiveness of reusing historical fuzzing results within continuous integration. To simulate this scenario, we selected four distinct programs, each with five different commits and versions, to ensure the observed discrepancies and generate historical fuzzing results. By reusing these fuzzing results, the newly generated corpus by MCM should be able to enhance fuzzing efficiency in the next commit or version, demonstrating its adaptability in a continuous integration workflow.

6.2. Experiment configuration

Platform. All the experiments were conducted on a 64-bit machine, and each fuzzing evaluation was executed with one CPU core of 2.40GHz E5-2640 V4. The operating system of the machine is Ubuntu 20.04 LTS. We run all target programs for 24 hours and repeat each experiment five times following the instruction of Klees et al. [23].

Target programs. We evaluate MCM-CMIN and MCM-OPT on 10 open-source Linux programs as shown in Table 3. The reason for choosing these programs is that most of them were evaluated by existing AFL-type fuzzers [23, 28, 32]. Among them, pdfimages, pdftotext, and libxml are executed with dictionaries provided by AFL++ [13].

Baseline. We compare MCM-CMIN and MCM-OPT with two state-of-art corpus minimization tools, OPTIMIN [19], AFL-CMIN [64], and with the original fuzzing results. Since all the OPTIMIN, MCM-OPT, and MCM-CMIN do not take into account edge frequency, we also use AFL-CMIN edge only (AFL-CMIN-E) as another baseline.

Historical fuzzing results. We executed the aforementioned 10 open-source programs on the same platforms for 24 hours. The initial corpus consisted of 20 seeds, which were collected from MOPT [32] or the Internet and deduplicated with AFL-CMIN, following the best practice. Every fuzzing process was repeated five times, and outputs obtained from these repetitions are treated as historical fuzzing results. Meanwhile, to overcome the limitations of the SAT solver, we initially simplify the result of each historical fuzzing result using OPTIMIN before conducting OPTIMIN and MCM-OPT.

Additionally, we selected four distinct program types, each with five different versions and five different commits, and subjected them to a 24-hour fuzzing process using the corresponding set of 20 initial seeds. This is done to validate the applicability of historical fuzzing results for programs with slight modifications.

Seed sets. Based on history fuzzing results, we use different corpus minimization tools to generate the initial seed corpus. Since we use the default corpus size of 100, we choose 80 seeds randomly from OPTIMIN and MCM-OPT since they treat every chosen seed equally. For the other three minimization methods, we choose the best 80 seeds from their generated corpus. All the corpus will incorporate the initial 20 seeds together as the seed corpus. Thus, each fuzzer will employ the same size of input files as the initial seeds corpus to fuzz the same program.

Evaluation metrics. We assess the performance of five corpus minimization methods from two metrics. The first metric pertains to branch coverage and corpus analysis, measuring the corpus generated by different minimization methods and their performance. Then vulnerability analysis is employed to ensure that all the crashes found in the historical fuzzing results are also present in the newly generated corpus.

6.3. Coverage analysis

As depicted in Figure 4 and Table 4, using historical fuzzing results to generate the initial corpus with different minimization tools can improve the fuzzing efficiency from 1% to 14.3% compared to the original fuzzing results. Besides, MCM-CMIN demonstrates superior performance compared to the other four minimization methods on 6 of 10 programs. Additionally, MCM-OPT outperforms other minimization tools in the case of nm and objdump. This is mainly in these two programs the seeds selected by MCM-OPT contain more edges whose appearance frequency is three, which contributes more to path exploration than edges whose appearance frequency is one and two. The appearance frequency of an edge represents the number of times this edge appears in different historical fuzzing campaigns. For libpng and pdfimages, both MCM-OPT and MCM-CMIN exhibit a slight disadvantage when compared to AFL-CMIN-E. Since the original coverage of libpng reaches the limit, all the fuzzing results using corpora generated by different minimization methods are very close to each other. In pdfimages, we suspect some special seeds can have a very large impact on the quality of the entire corpus, and the seeds may have little relation with edge appearance frequency.

Due to the random selection of seeds from the corpus generated by OPTIMIN, it produces a negative growth in comparison to the historical fuzzing results in four of the

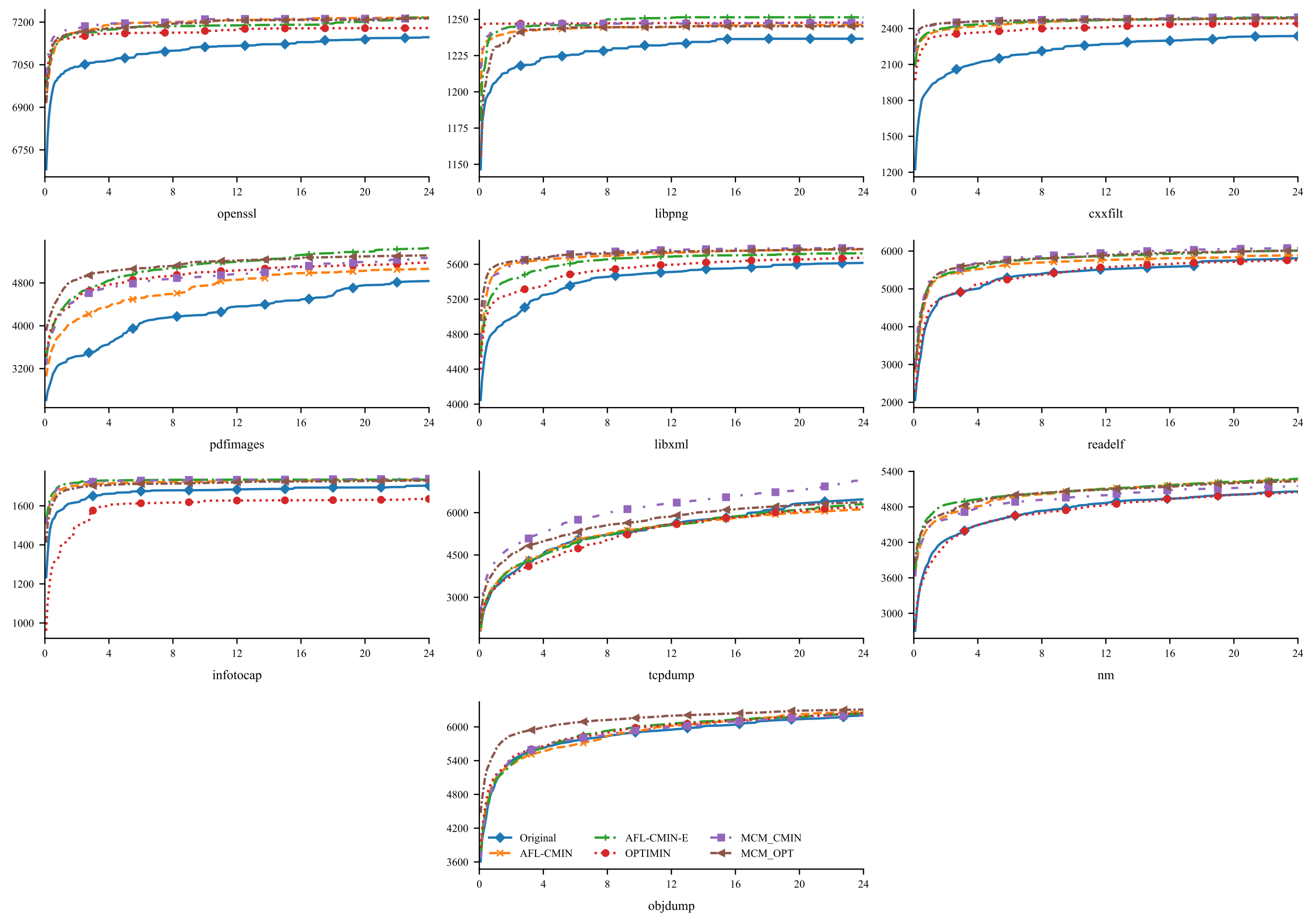


Fig. 4: Original branch coverage and average branch coverage that uses corpora generated by five different minimization methods over five campaigns in 24 hours.

programs. Meanwhile, since the rare edges are relatively scattered among different seeds in tcpdump, AFL-CMIN, and AFL-CMIN-E fail to select the most important part of the fuzzing results, leading to the decrease of corpus quality, therefore also producing a negative growth. MCM-CMIN excels in learning the relationship across different fuzzing campaigns, enabling it to select more efficient seeds compared to AFL-CMIN and AFL-CMIN-E, thus achieving a better performance. MCM-CMIN delivers similar or better performance in the other programs by learning the relationship across different fuzzing campaigns and selecting seeds that execute faster and appear later, thus reducing the additional cost to trigger special branches.

In summary, MCM-CMIN and MCM-OPT can select more efficient seeds from different fuzzing results, and the performance of MCM-CMIN and MCM-OPT is better than AFL-CMIN, AFL-CMIN-E, and OPTIMIN.

6.4. Vulnerabilities analysis

We evaluated the performance of different initial corpora generated by different minimization methods in discovering vulnerabilities. As is shown in Table 4, indicates that three programs experienced crashes in the historical fuzzing results. Since our research does not address runtime mutation strategies and seed selection, our focus lies in determining whether the different methods can identify vulnerabilities that align with or surpass the results obtained from historical data. The crashes observed in the historical fuzzing results can also be triggered by the new corpora generated by the different minimization tools.

We confirm that all of our found unique crashes have already been reported. For example, the crash of readelf has been reported by others as CVE-2021-20294 [37], which happens on line 12096 of $binutils/readelf.c$. The utilization of minimization tools enables the generation of an effective corpus since all the crashes' corresponding edges are rare. Thus ensuring that all previously discovered crashes will be triggered in subsequent fuzzing campaigns.

6.5. Initial corpus analysis

Table 5 displays the corpus size generated by five different minimization tools on 10 target programs when the corpus size is not limited to a given number. Due to the limitation of the SAT solver, the history results used by OPTIMIN and MCM-OPT are simplified before the minimization process. Meanwhile, since OPTIMIN and MCM-OPT use SAT solvers to get a globally optimal solution,

Table 4
The average branch coverage of historical fuzzing results referred to as Original branch coverage and average coverage of five corpus minimization methods running 24 hours on 10 target programs.

Targets	Original branch coverage & unique crashes	Branch coverage & unique crashes				
		OPTIMIN	AFL-CMIN	AFL-CMIN-E	MCM-OPT	MCM-CMIN
openssl	7147 / 0	7179 / 0	7216 / 0	7217 / 0	7213 / 0	**7238** / 0
libpng	1237 / 0	1248 / 0	1246 / 0	**1251** / 0	1245 / 0	1248 / 0
cxxfilt	2337 / 0	2441 / 0	2484 / 0	2493 / 0	2487 / 0	**2494** / 0
pdfimages	4819 / 1	5178 / 1	5064 / 1	**5453** / 1	5319 / 1	5265 / 1
libxml	5616 / 0	5676 / 0	5772 / 0	5728 / 0	5775 / 0	**5786** / 0
readelf	5802 / 1	5759 / 1	5886 / 1	6014 / 1	6014 / 1	**6081** / 1
infotocap	1700 / 1	1635 / 1	1733 / 1	1734 / 1	1729 / 1	**1739** / 1
tcpdump	6307 / 0	6236 / 0	6120 / 0	6302 / 0	6368 / 0	**7207** / 0
nm	5062 / 0	5048 / 0	5230 / 0	5234 / 0	**5238** / 0	5156 / 0
objdump	6204 / 0	6252 / 0	6271 / 0	6238 / 0	**6309** / 0	6230 / 0

Table 5
Corpus size minimized on historical results.

Target	ORIGINAL	OPTIMIN	AFL-CMIN	AFL-CMIN-E	MCM-OPT	MCM-CMIN
openssl	8760	240	776	249	31	309
libpng	7325	144	820	227	6	240
cxxfilt	43738	1799	4199	668	113	549
pdfimages	36959	862	5317	975	324	1196
libxml	45255	578	4225	885	122	938
readelf	89132	1028	9136	2158	222	1851
infotocap	27549	170	2225	331	29	342
tcpdump	38174	3318	7643	3954	878	4077
nm	65658	821	4288	1034	308	1166
objdump	80242	845	5016	1110	302	1266

the corpora generated by these two methods are relatively smaller.

Across all the minimization methods, the corpora generated by MCM-OPT are significantly smaller than other minimization methods. This can partly be attributed to the fact that four minimization methods except AFL-CMIN do not consider edge hit count in seed selection. Additionally, MCM-OPT prioritizes the least number of seeds that contain as many rare edges as possible, which further reduces the size of the corpus. Compared to OPTIMIN, MCM-OPT only considers the rare edges that appear in different campaigns and disregards the relatively frequent ones. As a result, MCM-OPT generates smaller corpora.

Since AFL-CMIN differentiates edges based on the edge hit count, its generated corpora are larger than the other four methods. When compared to AFL-CMIN-E, most of the corpora generated by MCM-CMIN are slightly larger. This can be attributed to the fact that MCM-CMIN takes into account the relation of edge appearance frequency across different fuzzing campaigns, which makes its corpus size a bit larger.

We further analyze the edge distribution of the corpora generated by different minimization methods when the corpus size is limited, as shown in Table 6, to learn the impact of edge appearance frequency on the corpus. Since OPTIMIN treats each seed equally and 80 seeds are randomly selected from the generated corpus, making corpus contains fewer rare edges than the other four methods. When comparing MCM-CMIN with AFL-CMIN and AFL-CMIN-E, it becomes evident that MCM-CMIN contains more rare edges whose edge appearance frequency is one. That is because MCM-CMIN prioritizes edges with smaller appearance frequency, thereby retaining seeds associated with these specific edges. Although MCM-OPT exhibits fewer edges with the appearance frequency of one, it preserves edge appearance frequency with a balanced distribution, encompassing more edges whose occurrences of two or three.

By analyzing the edge distribution of the generated corpus as well as their corresponding fuzz testing results, we can conclude that using edge appearance frequency to help seed selection from historical corpora can generate a high-quality initial corpus.

6.6. History results on slightly changed programs

To verify the efficiency of reusing historical fuzzing results in continuous integration, where the programs under testing are slightly changed in different fuzzing campaigns due to continuous software development, we selected four distinct programs that employ various input types. Additionally, we focused on five different commits or five consecutive versions to ensure the observed discrepancies and to simulate the continuously evolving programs. We obtained historical fuzzing results by using the initial 20 seeds to fuzz the programs of their five different commits or versions (as is shown in Table 8) each once. Thus, we still get five historical corpora for each target. Based on the generated historical fuzzing results, we followed the experiment configuration to get the corresponding seed sets. To verify the effectiveness of using historical fuzzing results, we chose the next commit or version as the target program and compared using the initial 20 seeds as a corpus with employing MCM's generated corpus.

The result is shown in Table 7, *Original* is the average branch coverage generated by using the initial 20 seeds. The performance of MCM-CMIN and MCM-OPT can effectively maintain branch coverage that aligns with the original corpus while exhibiting improvements. By MCM's using historical fuzzing results, noteworthy improvements were observed on tcpdump in terms of different commits, while in fields of different versions, better performance can be achieved on cxxfilt and pdfimages. That means leveraging

Table 6
The distribution of edges within the generated corpora. Each edge distribution comprises three columns, denoting edges that appear across distinct fuzzing campaigns with frequencies of one, two, and three, respectively.

Targets	Historical corpora distribution	New generated corpus distribution				
		OPTIMIN	AFL-CMIN	AFL-CMIN-E	MCM-OPT	MCM-CMIN
openssl	32 / 22 / 39	12 / 10 / 6	28 / 2 / 27	29 / 15 / 21	30 / 22 / 39	32 / 22 / 39
libpng	7 / 2 / 1	7 / 2 / 1	4 / 0 / 0	7 / 2 / 1	7 / 2 / 1	7 / 2 / 1
cxxfilt	29 / 156 / 85	0 / 7 / 6	28 / 60 / 13	29 / 100 / 33	18 / 131 / 63	29 / 141 / 16
pdfimages	387 / 262 / 401	9 / 31 / 165	99 / 26 / 0	132 / 31 / 47	79 / 74 / 232	173 / 30 / 46
libxml	94 / 97 / 80	14 / 4 / 1	79 / 37 / 33	45 / 28 / 46	55 / 69 / 47	94 / 74 / 33
readelf	101 / 120 / 274	0 / 0 / 0	83 / 8 / 52	72 / 44 / 35	18 / 51 / 111	101 / 13 / 63
infotocap	20 / 23 / 7	9 / 11 / 0	18 / 9 / 3	13 / 20 / 4	19 / 23 / 7	20 / 23 / 7
tcpdump	1464 / 1037 / 647	0 / 0 / 2	138 / 36 / 2	134 / 36 / 2	63 / 31 / 63	322 / 19 / 7
nm	341 / 231 / 179	0 / 0 / 1	148 / 46 / 15	119 / 66 / 17	65 / 41 / 45	221 / 59 / 22
objdump	213 / 171 / 201	0 / 2 / 0	129 / 23 / 29	128 / 58 / 34	35 / 19 / 52	157 / 30 / 31

Table 7
Branch coverage of history fuzzing result on new program version.

Target	Type	Original	MCM-CMIN		MCM-OPT	
			Coverage	Impr	Coverage	Impr
libpng-36bd1bb	C	849	850	+0.11%	850	+0.11%
openssl-5d91c74	C	6498	6594	+1.48%	6593	+1.46%
libxml-664f881	C	5628	5757	+2.29%	5774	+2.59%
tcpdump-211124b	C	6066	6901	+13.77%	6972	+14.94%
cxxfilt-2.36	V	3029	3190	+5.32%	3184	+5.12%
openssl-1.1.1j	V	6475	6599	+1.92%	6585	+1.68%
libpng-1.6.38	V	852	861	+1.06%	862	+1.17%
pdfimages-4.04	V	5330	5823	+9.25%	5343	+0.24%

fuzzing history to generate a new corpus ensures the performance of fuzzing in slightly changed programs. Therefore, utilizing historical results in programs with minor changes is practical.

7. Discussions and Limitations

7.1. Discussions

As demonstrated in the experiments, the corpus generated using historical fuzzing results can significantly improve fuzzing efficiency compared to the original corpus, increasing code coverage from 1% to 14.3%. This improvement is particularly notable when the edge needs to be triggered by different seeds, as using historical fuzzing results to create the initial corpus can greatly enhance coverage.

Although using historical fuzzing results can improve fuzzing efficiency, the choice of minimization tools plays a crucial role. Neither OPTIMIN nor AFL-CMIN can effectively learn relationships, and this limitation is addressed by MCM by computing edge appearance frequency across different fuzzing campaigns. Experiments show that MCM can retain more unique edges, enabling it to perform better than the other two tools. MCM enables users to save the results of each fuzzing campaign and reuse them in subsequent runs. Additionally, users can exchange fuzzing results with each other, using them as shared seeds for convenience.

We use slightly modified programs to simulate the OSS-FUZZ or CI scenario. The experiments show that using historical fuzzing results can lead to significant improvements, indicating that utilizing historical results in such CI scenarios can be practical.

7.2. Limitations

Since code coverage is just one indicator of fuzzing performance, our work still has some limitations. Many studies [5, 3, 12, 39] have suggested that there is a need to consider state coverage information. Since many vulnerabilities have a strong relation with specific states, only after both the edge and state meet the specific condition that the vulnerabilities can be triggered. Therefore, considering state information to aid in seed selection can effectively improve the quality of the initial corpus. After identifying some important states, we can extend MCM's seed selection factors to maintain these states to improve the corpus quality.

When reusing historical fuzzing results to generate the initial corpus for directed fuzzing [6, 30, 17], using just edge information to choose seeds is not enough. Since the purpose of directed fuzzing is to test specific code areas, considering the information of reaching a given set of target program locations is a greater alternative. We can extend MCM with taint analysis to find edges that correspond to specific program locations. After finding those special edges, MCM can further evaluate and preserve these edges to improve the relevance of the initial corpus to target program locations.

Additionally, we use slightly modified programs to simulate the CI scenario. The changes between different versions of the programs are quite small. However, if a user runs multiple fuzzing campaigns with different versions whose differences are quite large, reusing their results may not be effective, potentially threatening the validity of our work.

8. Related work

8.1. Initial seed corpus

Several papers [23, 31, 68, 19] underscore the crucial role of the initial corpus in determining the efficiency of fuzzing. A high-quality corpus effectively generates a broad range of observable behaviors in the target, reducing the effort required by the fuzzer to reach those areas. Enhancing

Table 8
The branch coverage of history program.

Type	Target	Branch Coverage				
Version	cxxfilt	2.33.1	2.34	2.35	2.35.1	2.35.2
		2277	2376	2326	2326	2444
	openssl	1.1.1e	1.1.1f	1.1.1g	1.1.1h	1.1.1i
		6527	6475	6465	6479	6576
	libpng	1.6.33	1.6.34	1.6.35	1.6.36	1.6.37
		850	851	851	851	866
	pdfimages	4.00	4.01	4.01.1	4.02	4.03
		4060	5117	4128	4901	5305
Commit	libpng	eb67672	6c6f7d1	c4bd411	763c77e	5f5f98a
		849	849	848	850	849
	openssl	d9d838	ba4c89a	32f7f60	9d86884	4f850d7
		6574	6518	6509	6486	6506
	libxml	c103566	5eeb9d5	5f1f455	0b79359	eee1dd5
		5580	5579	5637	5631	5676
	tcpdump	aa3e54f	12f66f6	6510207	9ba9138	af2cf04
		6371	5825	6179	6213	6300

the quality of the initial corpus can effectively improve the fuzz testing performance [19].

One approach to enhance the quality of the initial corpus involves integrating seed generators with other technologies, such as symbolic execution and machine learning algorithms. By utilizing symbolic execution, fuzzers can compute values that fulfill conditions or complex checks, thus improving the quality of seeds and enabling the detection of deeper code branches [51, 56, 42]. HEALER dynamically analyzes minimized test cases to establish relationships among them, leveraging these learned relations to guide input generation and improve the quality of test cases [52]. Lyu et al. proposed an unsupervised learning system named SmartSeed to learn and generate valuable input seed files for fuzzing [31]. Chen et al. used Recurrent Neural Networks to discover the correlation of specific pdf files with the target programs and further generate new seed files that more likely explore new paths [10]. Superion [54], SoFi [18] and PolyGlot [9] select JavaScript code crawled from Github repositories as seed corpus to drive the fuzzing process. Wen et al. evaluated the impact of seed selection on fuzzing effectiveness and found that the corpus with code features can lead fuzzers to achieve better results [60], which is consistent with our study. Zhi et al. conducted a systematic study to understand the impact of initial seed inputs on DL testing and proposed an SOO-based selection method to construct the optimal corpus, thereby enhancing DL testing with respect to specific testing goals [67].

The selection of a subset of inputs from a large collection corpus using various strategies is another approach to enhancing the quality of the initial corpus. Abdelnur et al. were the first to formally define the corpus minimization problem as an instance of the minimum set cover problem, and they employed a greedy algorithm to resolve this issue [1]. Rebert et al. proposed a method called MINSET, which generates a small subset considering factors such as execution time or file size [47]. AFL-CMIN utilizes AFL's edge coverage notion to select the smallest seed that covers a specific edge count from the collection corpus, following a greedy minimization approach [64]. MINTS [20] and Nemo [29] both use integer linear programming solvers to perform test-suite minimization. Additionally, OPTIMIN [19] encodes the subset selection as a maximum satisfiability problem and employs the EvalMaxSAT solver to address corpus minimization. Furthermore, MoonShine [41] uses system call traces of real-world programs to distill them into a minimal test case that still achieves 86 percent of the pre-distilled coverage.

MCM is orthogonal to the approaches that involve integrating seed generators with other technologies. It simply learns the edge relation across the different historical corpus to guide seed selection. Similar to existing works, MCM tries to select a more effective subset of the collected corpus to enhance the quality of the initial corpus, but it first proposes to consider the historical fuzzing results as the collected corpus to generate a high-quality corpus.

8.2. Using history information

In recent years, several studies have focused on leveraging historical information in the field of fuzzing. This historical data contains valuable pre-execution details that can be harnessed for further utilization. M Woo et al. proposed a simulation-based approach to replay the events of previous fuzzing and trained a better scheduling algorithm [61]. Lee et al. utilized historical datasets to create a neural network language model-guided fuzzer called Montage, capable of generating valid JavaScript tests [25]. Lyu et al. introduced a mutation model that captures mutation strategies from both intra- and inter-trial histories and presented a novel history-driven mutation framework known as EMS, designed to enhance mutation efficiency [33]. Furthermore, Rajpal et al. presented a machine-learning technique that employs neural networks to learn patterns in the input files from previous fuzzing explorations, thereby guiding future fuzzing endeavors [46]. Klooster et al. investigated the effectiveness of using continuous fuzzing in CI/CD pipelines [24] and CID-Fuzz [66] proposed using historical information in the CI process to perform differential analysis, addressing the issue of frequent code changes and improving fuzzing efficiency.

The techniques mentioned above share a common theme of utilizing historical fuzzing results. While our approach also incorporates historical results, we place greater emphasis on learning the relationships across different historical results to enable a more effective selection of meaningful seeds.

9. Conclusion

For programs deployed on the OSS-FUZZ or during the software integration development cycle, fuzz testing needs to be conducted multiple times. That means multiple historical fuzzing results are generated for every program, extracting useful seeds from these historical results can help explore the program more efficiently in later fuzz testing. In this paper, we propose to reuse historical fuzzing results and present a minimization tool called MCM to generate a high-quality corpus. MCM efficiently learns edge relations across different fuzzing campaigns and selects useful seeds from the historical fuzzing corpora. We have illustrated its effectiveness by comparing MCM with other state-of-the-art minimization tools and the original historical results. The results show that using historical fuzzing results can effectively improve the fuzzing performance.

References

[1] Abdelnur, H., Lucangeli, O.J., Festor, O., et al., 2010. Spectral fuzzing: Evaluation & feedback. Ph.D. thesis. INRIA.

[2] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J., 2017. Agile software development methods: Review and analysis. arXiv preprint arXiv:1709.08439 .

[3] Aschermann, C., Schumilo, S., Abbasi, A., Holz, T., 2020. Ijon: Exploring deep state spaces via fuzzing, in: 2020 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 1597–1612.

[4] Avellaneda, F., 2020. A short description of the solver evalmaxsat. MaxSAT Evaluation 8.

[5] Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A., 2022. Stateful greybox fuzzing, in: 31st USENIX Security Symposium (USENIX Security 22), USENIX Association, Boston, MA. pp. 3255–3272. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

[6] Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A., 2017. Directed greybox fuzzing, in: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp. 2329–2344.

[7] Cha, S.K., Woo, M., Brumley, D., 2015. Program-adaptive mutational fuzzing, in: 2015 IEEE Symposium on Security and Privacy, IEEE. pp. 725–741.

[8] Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., Liu, W., 2018. A systematic review of fuzzing techniques. Computers & Security 75, 118–137.

[9] Chen, Y., Zhong, R., Hu, H., Zhang, H., Yang, Y., Wu, D., Lee, W., 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation, in: 2021 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 642–658.

[10] Cheng, L., Zhang, Y., Zhang, Y., Wu, C., Li, Z., Fu, Y., Li, H., 2019. Optimizing seed inputs in fuzzing with machine learning, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE. pp. 244–245.

[11] De Moura, L., Bjørner, N., 2008. Z3: An efficient smt solver, in: International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer. pp. 337–340.

[12] Fioraldi, A., D'Elia, D.C., Balzarotti, D., 2021. The use of likely invariants as feedback for fuzzers, in: 30th USENIX Security Symposium (USENIX Security 21), pp. 2829–2846.

[13] Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M., 2020. {AFL++}: Combining incremental steps of fuzzing research, in: 14th USENIX Workshop on Offensive Technologies (WOOT 20).

[14] Fowler, M., Foemmel, M., 2006. Continuous integration.

[15] Gan, S., Zhang, C., Chen, P., Zhao, B., Qin, X., Wu, D., Chen, Z., 2020. {GREYONE}: Data flow sensitive fuzzing, in: 29th USENIX security symposium (USENIX Security 20), pp. 2577–2594.

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

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

[18] He, X., Xie, X., Li, Y., Sun, J., Li, F., Zou, W., Liu, Y., Yu, L., Zhou, J., Shi, W., et al., 2021. Sofi: Reflection-augmented fuzzing for javascript engines, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 2229–2242.

[19] Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., Hosking, A.L., 2021. Seed selection for successful fuzzing, in: Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis, pp. 230–243.

[20] Hsu, H.Y., Orso, A., 2009. Mints: A general framework and tool for supporting test-suite minimization, in: 2009 IEEE 31st international conference on software engineering, IEEE. pp. 419–429.

[21] Jauernig, P., Jakobovic, D., Picek, S., Stapf, E., Sadeghi, A.R., 2022. Darwin: Survival of the fittest fuzzing mutators. arXiv preprint arXiv:2210.11783 .

[22] Kaelbling, L.P., Littman, M.L., Moore, A.W., 1996. Reinforcement learning: A survey. Journal of artificial intelligence research 4, 237–285.

[23] Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M., 2018. Evaluating fuzz testing, in: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp. 2123–2138.

[24] Klooster, T., Turkmen, F., Broenink, G., Ten Hove, R., Böhme, M., 2023. Continuous fuzzing: a study of the effectiveness and scalability of fuzzing in ci/cd pipelines, in: 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), IEEE. pp. 25–32.

[25] Lee, S., Han, H., Cha, S.K., Son, S., 2020. Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer, in: 29th USENIX Security Symposium (USENIX Security 20), pp. 2613–2630.

[26] Li, C.M., Manya, F., 2021. Maxsat, hard and soft constraints, in: Handbook of satisfiability. IOS Press, pp. 903–927.

[27] Li, J., Li, S., Sun, G., Chen, T., Yu, H., 2022. Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. IEEE Transactions on Information Forensics and Security 17, 2673–2687.

[28] Li, Y., Ji, S., Chen, Y., Liang, S., Lee, W.H., Chen, Y., Lyu, C., Wu, C., Beyah, R., Cheng, P., et al., 2021. {UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers, in: 30th USENIX Security Symposium (USENIX Security 21), pp. 2777–2794.

[29] Lin, J.W., Jabbarvand, R., Garcia, J., Malek, S., 2018. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming, in: Proceedings of the 40th International Conference on Software

Engineering, pp. 1039–1049.

[30] Luo, C., Meng, W., Li, P., 2023. Selectfuzz: Efficient directed fuzzing with selective path exploration, in: 2023 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 2693–2707.

[31] Lyu, C., Ji, S., Li, Y., Zhou, J., Chen, J., Chen, J., 2018. Smartseed: Smart seed generation for efficient fuzzing. arXiv:arXiv:1807.02606.

[32] Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.H., Song, Y., Beyah, R., 2019. {MOPT}: Optimized mutation scheduling for fuzzers, in: 28th USENIX Security Symposium (USENIX Security 19), pp. 1949–1966.

[33] Lyu, C., Ji, S., Zhang, X., Liang, H., Zhao, B., Lu, K., Beyah, R., 2022. Ems: History-driven mutation for coverage-based fuzzing, in: 29th Annual Network and Distributed System Security Symposium. https://dx. doi. org/10.14722/ndss.

[34] Mallissery, S., Wu, Y.S., 2023. Demystify the fuzzing methods: A comprehensive survey. ACM Computing Surveys 56, 1–38.

[35] Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M., 2019. The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering 47, 2312–2331.

[36] Metzman, J., Szekeres, L., Simon, L., Sprabery, R., Arya, A., 2021. Fuzzbench: an open fuzzer benchmarking platform and service, in: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp. 1393–1403.

[37] Modra, 2020. Cve-2021-20294 patch. https://sourceware.org/bugzilla/show_bug.cgi?id=26929.

[38] Mozilla, 2020. Fuzzing-test samples. https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html.

[39] Natella, R., 2022. Stateafl: Greybox fuzzing for stateful network servers. Empirical Software Engineering 27, 191.

[40] Natella, R., Pham, V.T., 2021. Profuzzbench: A benchmark for stateful protocol fuzzing, in: Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis, pp. 662–665.

[41] Pailoor, S., Aday, A., Jana, S., 2018. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation, in: 27th USENIX Security Symposium (USENIX Security 18), pp. 729–743.

[42] Pak, B.S., 2012. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. School of Computer Science Carnegie Mellon University .

[43] Peng, H., Shoshitaishvili, Y., Payer, M., 2018. T-fuzz: fuzzing by program transformation, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 697–710.

[44] Petsios, T., Tang, A., Stolfo, S., Keromytis, A.D., Jana, S., 2017. Nezha: Efficient domain-independent differential testing, in: 2017 IEEE Symposium on security and privacy (SP), IEEE. pp. 615–632.

[45] Qin, S., Hu, F., Ma, Z., Zhao, B., Yin, T., Zhang, C., 2023. Nsfuzz: Towards efficient and state-aware network service fuzzing. ACM Transactions on Software Engineering and Methodology .

[46] Rajpal, M., Blum, W., Singh, R., 2017. Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596 .

[47] Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D., 2014. Optimizing seed selection for fuzzing, in: 23rd USENIX Security Symposium (USENIX Security 14), pp. 861–875.

[48] Schumilo, S., Aschermann, C., Jemmett, A., Abbasi, A., Holz, T., 2022. Nyx-net: network fuzzing with incremental snapshots, in: Proceedings of the Seventeenth European Conference on Computer Systems, pp. 166–180.

[49] Serebryany, K., 2017. OSS-Fuzz - google's continuous fuzzing service for open source software, USENIX Association, Vancouver, BC.

[50] Shahrokni, A., Feldt, R., 2013. A systematic review of software robustness. Information and Software Technology 55, 1–17.

[51] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution., in: NDSS, pp. 1–16.

[52] Sun, H., Shen, Y., Wang, C., Liu, J., Jiang, Y., Chen, T., Cui, A., 2021. Healer: Relation learning guided kernel fuzzing, in: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pp. 344–358.

[53] Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire: Data-driven seed generation for fuzzing, in: 2017 IEEE Symposium on Security and Privacy (SP), IEEE. pp. 579–594.

[54] Wang, J., Chen, B., Wei, L., Liu, Y., 2019. Superion: Grammar-aware greybox fuzzing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 724–735.

[55] Wang, J., Zhang, Z., Liu, S., Du, X., Chen, J., 2023. Fuzzjit: Oracle-enhanced fuzzing for javascript engine jit compiler, in: USENIX Security Symposium. USENIX.

[56] Wang, T., Wei, T., Gu, G., Zou, W., 2011. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. ACM Transactions on Information and System Security (TISSEC) 14, 1–28.

[57] Wang, W., Sun, H., Zeng, Q., 2016. Seededfuzz: Selecting and generating seeds for directed fuzzing, in: 2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE. pp. 49–56.

[58] Wang, Y., Jia, X., Liu, Y., Zeng, K., Bao, T., Wu, D., Su, P., 2020. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization., in: NDSS.

[59] Weiss, K., Khoshgoftaar, T.M., Wang, D., 2016. A survey of transfer learning. Journal of Big data 3, 1–40.

[60] Wen, M., Wang, Y., Xia, Y., Jin, H., 2023. Evaluating seed selection for fuzzing javascript engines. Empirical Software Engineering 28, 133.

[61] Woo, M., Cha, S.K., Gottlieb, S., Brumley, D., 2013. Scheduling black-box mutational fuzzing, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 511–522.

[62] Xu, W., Kashyap, S., Min, C., Kim, T., 2017. Designing new operating primitives to improve fuzzing performance, in: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp. 2313–2328.

[63] Ye, A., Wang, L., Zhao, L., Ke, J., Wang, W., Liu, Q., 2021. Rapidfuzz: Accelerating fuzzing via generative adversarial networks. Neurocomputing 460, 195–204.

[64] Zalewski, M., 2014. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[65] Zeng, Y., Zhu, F., Zhang, S., Yang, Y., Yi, S., Pan, Y., Xie, G., Wu, T., 2023. Dafuzz: data-aware fuzzing of in-memory data stores. PeerJ Computer Science 9, e1592.

[66] Zhang, J., Cui, Z., Chen, X., Yang, H., Zheng, L., Liu, J., 2023. Cidfuzz: Fuzz testing for continuous integration. IET Software 17, 301–315.

[67] Zhi, Y., Xie, X., Shen, C., Sun, J., Zhang, X., Guan, X., 2023. Seed selection for testing deep neural networks. ACM Transactions on Software Engineering and Methodology 33, 1–33.

[68] Zhu, X., Wen, S., Camtepe, S., Xiang, Y., 2022. Fuzzing: a survey for roadmap. ACM Computing Surveys (CSUR) 54, 1–36.