Journal of King Saud University - Computer and Information Sciences 36 (2024) 101920

HOSTED BY

Contents lists available at ScienceDirect

Journal of King Saud University - Computer and Information Sciences

journal homepage: www.sciencedirect.com

Full length article

Grammar-aware test case trimming for efficient hybrid fuzzing ☆

Yiru Zhao, Long Gao, Qihan Wan, Lei Zhao [1]

Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan, Hubei, 430072, China

ARTICLE INFO

Keywords:
Hybrid fuzzing
Fuzzing
Concolic execution
Delta debugging

ABSTRACT

In recent years, hybrid fuzzing has garnered significant attention by combining the benefits of both fuzzing and concolic execution. However, existing hybrid fuzzing techniques face problems such as the performance overhead of concolic execution and low code coverage in real-world programs. In this study, we observed that concolic execution often falls into repetitive loop structures when analyzing inputs generated by fuzzing, leading to the construction of redundant constraint expressions, which severely impacts its efficiency. Based on this observation, we design a test case trimming approach to remove input fields related to repetitive loop structures. We propose a lightweight taint analysis technique to infer input's grammatical structure and construct a hierarchical grammar tree. Then, through delta debugging, we identify nodes that do not affect the code coverage and remove the corresponding input fields. We implement a prototype *ScissorFuzz*. Experimental results show that our approach brings an average of 8.2% improvement in code coverage and triggers 115 more crashes compared to the state-of-the-art hybrid fuzzing technique QSYM. By eliminating redundant fields, our approach reduces the resource and time overhead associated with concolic execution during input processing, thereby enhancing the efficiency of hybrid fuzzing.

1. Introduction

Software vulnerability detection has always been a research focus in both academic and industry communities. There are two mainstream automated vulnerability detection techniques, namely fuzzing (Zalewski, 2015; honggfuzz, 2010) and concolic execution (Chipounov et al., 2011). Fuzzing enables rapid exploration of the program space through random mutations but faces challenges when encountering complex path constraints, such as magic bytes. On the other hand, concolic execution can cover program branches with complex constraints, but it is time-consuming and requires significant resources.

In order to leverage the respective strengths of fuzzing and concolic execution, hybrid fuzzing (Stephens et al., 2016) has been proposed. Hybrid fuzzing utilizes fuzzing to quickly explore branches in the program with loose constraints, while simultaneously employing concolic execution to assist in exploring branches with complex constraints. As depicted in Fig. 1, most existing hybrid fuzzing approaches employ a cooperative mechanism. Specifically, during the execution of fuzzing, test cases generated by fuzzing are selectively employed to launch concolic execution. Subsequently, the test cases generated by concolic execution are fed back to fuzzing. This iterative process enables hybrid fuzzing to dynamically alternate between fuzzing and concolic execution, facilitating comprehensive exploration of the program's paths.

In recent years, several techniques have been proposed to enhance the performance of hybrid fuzzing. For instance, QSYM (Yun et al., 2018) employs optimization techniques to accelerate the execution speed of concolic execution. DigFuzz (Zhao et al., 2019) focuses on seed scheduling strategies, which employ various metrics to select the most valuable test cases generated by fuzzing to enable concolic execution to prioritize the execution of them. Intriguer (Cho et al., 2019) optimizes the construction of constraint expressions to enhance the efficiency of concolic execution.

☆ Fundings: This work was supported by National Natural Science Foundation of China [No.62172305], Hubei Province Key Research and Development Program [No.2021BAA027].

E-mail addresses: zhaoyiru@whu.edu.cn (Y. Zhao), longgao@whu.edu.cn (L. Gao), wanqihan@whu.edu.cn (Q. Wan), leizhao@whu.edu.cn (L. Zhao).
[1] Corresponding author: Lei Zhao.

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2024.101920
Received 30 October 2023; Received in revised form 19 December 2023; Accepted 6 January 2024
Available online 9 January 2024

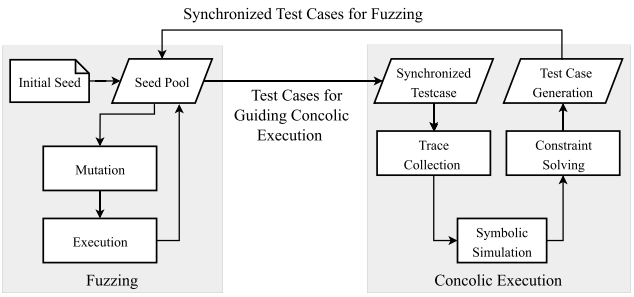


Fig. 1. Overview of hybrid fuzzing.

Despite various optimizations being proposed, the improvement of hybrid fuzzing remains relatively modest in real-world programs. One of the reasons is that the execution speed of concolic execution is persistently slow. This can be attributed to several factors. Firstly, the process of concolic execution necessitates the translation of binary instructions into an intermediate language and subsequent interpretation. Moreover, during the execution, a substantial number of constraint expressions must be generated and managed, which requires a lot of computational resources. Consequently, existing research endeavors have concentrated on enhancing the efficiency of concolic execution. For instance, SymCC (Poeplau and Francillon, 2020) incorporates the compilation-time collection of constraint code embedded within the program, resulting in concolic execution with execution speeds approaching that of native execution. Furthermore, SymSan (Chen et al., 2022) exploits highly optimized data flow analysis frameworks, thereby substantially mitigating the overhead associated with constraint expressions.

However, existing studies primarily focus on concolic execution itself and ignore the impact of the cooperative mechanism employed in hybrid fuzzing on its performance. We observed that test cases passed from fuzzing to concolic execution often result in concolic execution getting trapped in redundant program loops. These redundant loops not only fail to facilitate the discovery of new paths but also significantly impede the execution speed of concolic execution. This is primarily attributed to excessive statement simulation and the construction of unnecessary constraint expressions. Therefore, eliminating the fields in test cases that contribute to redundant loops not only does not affect the path exploration of concolic execution but also improves efficiency. In this paper, we define such fields as *redundant fields* in test cases.

To trim *redundant fields*, it is necessary to obtain the grammatical structure of test cases based on the processing logic of the target program and identify the fields corresponding to program loops. To get the processing logic of the target program, a direct approach is to employ taint analysis, traditional taint analysis techniques introduce overhead due to fine-grained tracking of each instruction. In this study, we propose a lightweight taint analysis technique that simplifies the taint propagation logic of traditional techniques. Our observations indicate that programs typically adhere to a "parse before process" logic, wherein the program adheres to the logic of parsing the input before further processing. As our focus lies on grammatical structure, we only track the propagation of input copies in memory and cease tracking once the field has undergone deformation. Additionally, we introduce hierarchical taint labels as a replacement for the traditional taint labels that impose limitations on data length. These hierarchical taint labels are employed to describe the hierarchical grammatical structure of the input. By employing this lightweight taint analysis technique, we derive a grammar tree for the test case.

Based on the obtained grammar tree, we proceed with identifying and trimming of the *redundant fields*. Firstly, we perform static analysis on the nodes of the grammatical structure tree to quickly exclude most of the nodes that cannot be trimmed. Then, we employ a multi-level delta debugging algorithm to trim the nodes from the root node of the grammar tree in a top-down manner, progressively trimming the tree at each level. During this dynamic trimming process, we ensure that the removal of data segments corresponding to nodes does not alter the paths covered by test cases. Ultimately, the trimmed test cases will be passed to the concolic execution and fuzzing.

The approach we propose entails the lightweight identification and removal of redundant fields within test cases. This alleviates the efficiency issues overlooked by existing hybrid fuzzing techniques caused by redundant fields, thereby reducing the resource and time consumption associated with concolic execution and fuzzing in exploring the execution paths of programs. We implement a prototype *ScissorFuzz* based on our approach. To demonstrate the effectiveness, we select 6 highly structured applications from the real-world program dataset of UNIFUZZ (Li et al., 2021) as evaluation benchmarks. Experimental results demonstrate that, in terms of code coverage, our technique achieves an average improvement of 8.2% compared to QSYM. *ScissorFuzz* outperforms DigFuzz on all the programs. Regarding the number of crashes, QSYM detects 374 crashes, while our technique detects 489 crashes. *ScissorFuzz* finds 134 more crashes than DigFuzz. In terms of solving efficiency, the concolic executor of our technique generates more test cases on average than QSYM, indicating that *ScissorFuzz* has higher efficiency in concolic execution.

This study makes the following contributions:

- **New observation in hybrid fuzzing.** We present a new observation that random mutations of fuzzing often result in low-quality test cases which contain redundant fields. More important, these redundant fields will further introduce negative impact on the efficiency or resource consumption of concolic execution. Based on this observation, we demonstrate the desirability of trimming redundant fields in test cases for hybrid fuzzing.
- **Lightweight dynamic taint tracking for test case trimming.** Trimming test cases relies on input formats and grammar. However, recognizing and identifying input fields is a time-consuming process. To address this problem, we propose a lightweight dynamic taint tracking technique that only tracks data movement instructions, enabling fast and accurate test case trimming.
- **New technique for hybrid fuzzing.** We design and implement our test case trimming technique in hybrid fuzzing. Experimental results show that our technique outperforms baseline techniques in terms of code coverage, vulnerability detection, and the efficiency of concolic execution.

2. Related work

In this section, we introduce related works of fuzzing and hybrid fuzzing.

2.1. Fuzzing

Fuzzing becomes popular especially since AFL has shown its effectiveness. Due to the difficulty of generating inputs that conform to the grammar specifications of test cases using random mutation, researchers propose various techniques to generate high-quality inputs in order to mitigate the impact of random mutation on fuzzing.

Alsaedi et al. (2022) propose a black-box fuzzing approach tailored for modern web applications. The approach involves the generation and mutation of input data based on the analysis of web application structure, input fields, and expected data formats. SCIPOG (Aminu Muazu et al., 2023) predefines certain values for specific parameters to ensure that important combinations are included in the generated test cases and integrates constraints to guide the generation process and enforce specific requirements or limitations. There are also heavyweight fuzzing techniques proposed that combine techniques such as taint analysis and dynamic instrumentation. For instance, Steelix (Li et al.,

2017) guides the mutation of fuzzing by identifying byte comparison information through static analysis. RedQueen (Liow et al., 2011) hooks comparison instructions and function calls, dynamically recording compared values. Angora (Chen and Chen, 2018) utilizes taint tracking to determine relevant byte offsets for guiding the mutations. GreyOne identifies critical bytes in program source code that need to be mutated through inference-based taint analysis, and performs targeted mutations on them (Gan et al., 2020).

To mitigate the computational cost of mutation, Hosseini et al. (2021) encode the mutation operators and error propagation paths into a chromosome representation, which is then subjected to a genetic algorithm-based optimization process to identify a subset of mutants that maximizes the fault coverage while minimizing the computational cost of the mutation testing process. Nasrin Shomali and Bahman Arasteh (Shomali and Arasteh, 2020) leverage a firefly optimization algorithm to guide the selection of high-quality mutants while discarding redundant or low-quality ones.

2.2. Hybrid fuzzing

The concept of hybrid fuzzing is first proposed by Majumdar and Sen (Majumdar and Sen, 2007). Hybrid fuzzing utilizes the constraint-solving capability of concolic execution to generate test cases that satisfy program path constraints. Concolic execution is a path-exploring technique that performs symbolic execution along a concrete execution path to direct the program to new execution paths.

In recent years, researchers focused on improving the efficiency of hybrid fuzzing through the following three aspects. First, many researchers focus on optimizing the execution speed of concolic execution. For example, QSYM discards useless features in concolic execution and designs a lightweight concolic execution technique for hybrid fuzzing. Intriguer only invokes the solver when the constraint expressions are particularly complex, thus reducing the burden of concolic execution. SymQemu and SymCC incorporate symbol collection code during compilation, greatly accelerating the execution speed of concolic execution. Pangolin (Huang et al., 2020), LeanSym (Mi et al., 2021), and SymSan aim to minimize or improve the generation and management of constraint expressions to accelerate the solving speed of concolic execution.

Second, several studies address the resource consumption issue of concolic execution by proposing various seed scheduling algorithms to select optimal test cases for concolic execution. MDPC (Lin et al., 2021) utilizes an optimal strategy based on Markov chains to evaluate the cost of each path for fuzzing and concolic execution, facilitating path allocation. DigFuzz (Zhao et al., 2019) employs a Monte Carlo method to assess the difficulty of solving a branch for fuzzing and passes the corresponding test cases for paths that are difficult to cover to concolic execution. MEUZZ (Chen et al., 2020) employs machine learning to learn a series of features of test cases and selects the valuable test cases to launch concolic execution.

Third, there is another class of hybrid fuzzing techniques that utilize symbolic execution to assist in generating initial test cases for fuzzing. This approach helps fuzzing bypass some initial program validation checks, enabling rapid exploration of internal program regions. TaintScope (Wang et al., 2010) employs taint analysis to identify relevant bytes, and then uses concolic execution to solve these bytes and generate initial test cases for fuzzing. CSEFuzz (Xie et al., 2020) initially employs static program exploration through symbolic execution to generate candidate test cases, and then selects a subset of test cases from the candidate pool based on a test case template and feeds them into the fuzzing.

3. Background and motivation

In this section, we first demonstrate the presence of redundant fields within the test cases through experimentation. Subsequently, we analyze the impact of these redundant fields on hybrid fuzzing. Finally, we discuss the limitations of existing techniques for redundancy removal and present our motivation in addressing this issue.

Fig. 2. Execution flow of mp42aac.

Table 1
Size and percentage of repetitive structures.

File type	File name	Repetitive structure		Nonrepetitive structure	
		Size (Byte)	Percentage	Size (Byte)	Percentage
mp4	1.mp4	30998	96.68%	1065	3.32%
	2.mp4	30632	96.53%	1101	3.47%
	3.mp4	30909	96.38%	1161	3.62%
jpg	2.jpg	17663	96.58%	625	3.42%
	4.jpg	21724	97.20%	625	2.80%
	9.jpg	14743	95.93%	625	4.07%
pdf	W.pdf	12489	98.21%	227	1.79%
	UTF8.pdf	13727	99.73%	37	0.27%
	BZ7.pdf	10966	96.98%	342	3.02%

3.1. Redundant fields in real-world programs

Existing techniques ignore the influence of the test case as input on concolic execution. We observed that test cases processed by programs exhibit a considerable presence of repetitive structural fragments. In these instances, the program consistently executes identical code within a loop when encountering these repetitive structural fragments. Fig. 2 illustrates a motivating example of mp42aac, which extracts audio from MP4 files. The main body of the program utilizes a loop to read a specific format of data fragments from the input and calls the handle function to process each data fragment. Within the handle function, mp42aac directly writes the data into the AAC file, and the execution path does not change based on the content of the object data. Therefore, mp42aac exhibits the same execution path when processing different data fragments. The *redundant fields* fragments have no impact on the test effectiveness of mp42aac. We define these data fragments that increase the number of loop iterations without discovering new paths or new program states as *redundant fields* in test cases.

Redundant fields are commonly found in highly structured test cases that exhibit repetitive structures. In various file types such as audio, video, prediction, and PDF, repetitive structures carry the primary content of the respective file types, such as video frame data, image pixel data, PDF object data, etc. We conducted a statistical analysis to calculate the proportion of repetitive structures in highly structured test cases. Specifically, we selected test cases of different file types from the initial test cases provided by the UniFuzz dataset (Li et al., 2021). By analyzing the execution processes of the test case, we counted the number of bytes in the test case that do not affect the program's execution path. The results are presented in Table 1. It can be observed that the proportion of data occupied by repetitive structures is significantly larger than other parts of the file.

3.2. Impact of redundant fields

Redundant fields significantly impact the efficiency of hybrid fuzzing. For fuzzing, *redundant fields* in test cases lead to a high number of irrelevant data hits during mutation, reducing the probability of generating interesting test cases after mutation. And for concolic execution, *redundant fields* in test cases result in the simulation of redundant instruction fragments during concolic execution, leading to the establishment and management of a large number of unnecessary constraint expressions.

To demonstrate the specific impact of *redundant fields* on the efficiency of concolic execution, we select a program, namely KPRC-

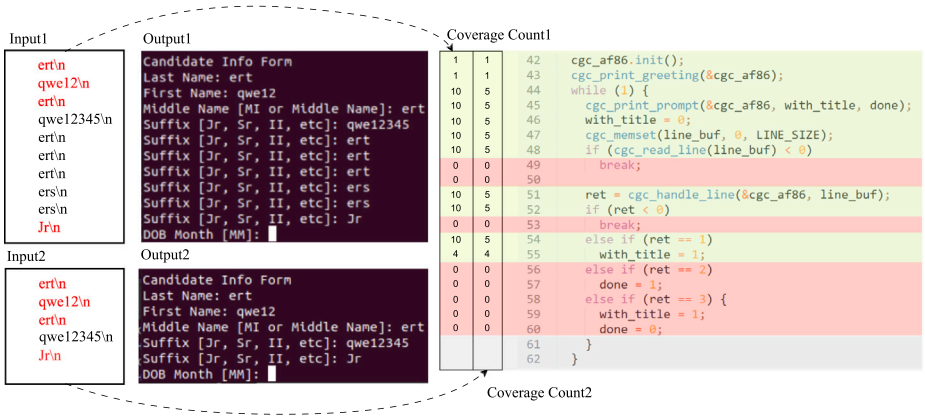


Fig. 3. *Redundant fields* of KPRCA00023.

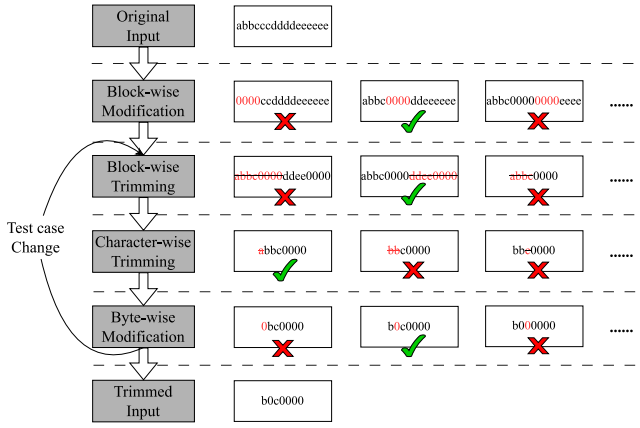


Fig. 4. AFL-tmin's test case trimming process.

A00023, which is one of the challenges in the "Cyber Grand Challenge" (CGC) organized by the Defense Advanced Research Projects Agency (DARPA) of the United States Department of Defense (DARPA, 2017). The program simulates an online application form where users need to continuously submit answers to the form's questions until all information is entered correctly. If a user's input fails validation, the program will repeatedly prompt the same question until the information is successfully entered.

For this program, in the two inputs shown on the left side of Fig. 3, the red highlighted portions represent correct inputs that can proceed to the next question based on the current validation and the black portions of the input fail validation. By observing the code coverage on the right side of the figure, it can be observed that the execution count increases for lines 45–48, 51–52, and 54. However, these additional executions do not result in any change of program state or increase in coverage. Therefore, the black portions represent *redundant fields* in the test case for this program.

Based on these characteristics, we manually construct a set of test cases with varying sizes but an equal number of valid answer submissions. We then use QSYM to solve each test case. The results are presented in Table 2, showing that as test cases grow larger, the time and memory usage for concolic execution also increase. And although concolic execution generates more new test cases, the coverage paths remain completely identical.

We observed that as fuzzing runs, there is a tendency for test case sizes to increase, accompanied by a gradual accumulation of *redundant fields*. This is primarily due to the random modifications made by fuzzing, it often incorporates mutation algorithms that introduce variations leading to an increase in input size. For example, AFL employs

Table 2
Cost of concolic execution for different test case sizes.

Input size (Byte)	Solve time (s)	Memory (MB)	Test case count
46	9	7.7	190
271	10	7.85	300
542	10	7.94	324
1.1K	11	8.01	330
2.6K	11	8.09	342
12.3K	21	8.61	356
29.1K	71	28.04	402
116.8K	175	36.37	428

the splice mutation operator, which merges two test cases into one. This operator has a higher disruptive effect on test cases, increasing the likelihood of altering coverage paths. As the duration of fuzzing extends, this mutation approach is employed more frequently, resulting in an overall growth in test case sizes throughout the fuzzing process and consequently generating a greater amount of *redundant fields*.

3.3. Existing test case trimming technique

Redundant fields affect the efficiency of both fuzzing and concolic execution, and its impact can be mitigated by trimming. However, it is important to note that the trimming process itself incurs additional overhead, which can also impact the overall efficiency. AFL (Zalewski, 2015) provides a tool called AFL-tmin afl-tmin (2013), which randomly trims data segments in a test case and determines whether the trims are effective by comparing the execution paths before and after the trims. As shown in Fig. 4, the trimming process consists of four stages. Firstly, AFL-tmin selects an appropriate step size based on the size of the test case. It then divides the test case into several data blocks according to this step size and iteratively fills the content of each data block with "\0" or removes the entire data block. Secondly, AFL-tmin collects all the characters that appear in the test case and attempts to remove each character individually. Finally, AFL-tmin replaces each byte of the data with "\0".

However, this completely random trimming approach disregards the grammar structure of the input, which contributes to a high failure rate. Moreover, in order to remove useless data as much as possible, the random trimming process is time-consuming. When using AFL-tmin to trim 1214 test cases generated by fuzzing for 18 programs, more than 90% of the test case trimming time exceeds 5 h, and the average success rate varies from 11.7% to 52.8% across different programs.

Some existing studies focus on addressing overlaps among test cases, aiming to reduce their quantity and alleviate redundancy. For example, BOOMPizer (Barisal et al., 2022) reduces the redundancy and overlap among test cases while preserving the coverage criteria. HDD (Misherghi and Su, 2006) leverages delta debugging algorithm to

Table 3
Instruction types for data flow propagation and their corresponding processing methods.

Instruction type	Flow direction	Instruction example	Process method
System call	FD2M	SYSCALL	Generate root-level taint labels
Data movement	R2M/M2R/R2R	MOV	Record temporary labels in registers
	M2M	MOVS/LODSx/STOSx	Direct propagation of taint labels
Data transformation	–	ADD/MUL/DIV/AND/XOR	Clear taint labels

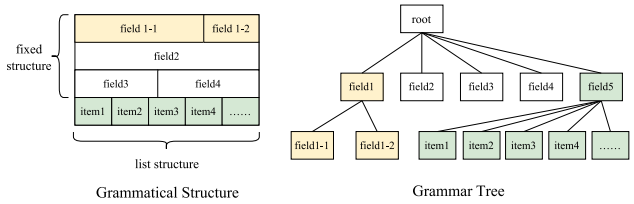


Fig. 5. Grammatical structure of test cases.

remove fields that are aligned with syntactic unit boundaries to lower the number of syntactically broken intermediate test cases. However, the internal redundancy within individual test cases, manifested by redundant fields, remains unaddressed and cannot be eliminated by these approaches. Therefore, a grammar-aware test case trimming technique is urgently needed.

3.4. Motivation

We observed that *redundant fields* are closely related to the grammatical structure of test cases. Analyzing the grammatical structural features of test cases can help identify *redundant fields*, enabling targeted trimming of specific data segments within the test case. As illustrated in Fig. 5, the grammatical structure of test cases can generally be categorized into fixed structures and list structures. A fixed structure refers to a configuration where each data field carries different semantics, and the number of segments is usually non-expandable. On the other hand, a list structure is composed of multiple identical semantic segments, and the number of segments can increase or decrease. By representing the grammatical structure of test cases using a tree-like structure as shown on the right side, we define it as a grammar tree.

Furthermore, the data segments processed by loop structures in the program often correspond to list structures in the grammatical structure. Each list item in the list structure exhibits the same semantics, and the number of list items is variable, without affecting the overall format of the test case. For such structural segments, the program typically needs to iterate through each list item for processing. The beginning of the iteration reads the data of the list item, and the specific content of the data determines the execution path of the current iteration. If two list item segments share certain similar characteristics, it may lead to identical execution paths within the two iterations. Thus, it can be assumed that *redundant fields* often exist within these list items. Therefore, our key idea lies in reconstructing the grammatical structure of test cases, identifying the list items, and removing *redundant fields* within them.

4. ScissorFuzz design

ScissorFuzz aims to improve the mutation efficiency of fuzzing and the execution efficiency of concolic execution by trimming test cases without compromising the path coverage. Fig. 6 shows an overview of *ScissorFuzz*'s architecture. To accomplish its objectives, *ScissorFuzz* incorporates an additional phase within the test case scheduling process, transitioning from fuzzing to concolic execution. During this phase, *ScissorFuzz* initially conducts a lightweight taint analysis on test cases to reconstruct their grammatical structure, culminating in the construction of a grammar tree. Subsequently, leveraging the grammar tree, *ScissorFuzz* employs delta debugging techniques to trim test cases. These trimmed test cases are then synchronized with the concolic execution for further analysis and execution. This synchronization enables concolic execution to achieve accelerated execution speed and generate more concise feedback on test cases for fuzzing. Consequently, the hit rate of fuzzing on critical bytes is amplified, thereby ultimately enhancing the overall efficiency of hybrid fuzzing.

4.1. Grammatical structure reconstruction

The grammatical structure of test cases holds a significant influence over the execution flow of a program during the processing of its input. This influence stems from the "parse before process" principle, wherein the program adheres to the logic of parsing the input before further processing. Leveraging this principle, we propose a lightweight taint analysis approach that concentrates on tracking data movement instructions (such as MOV, STOS, READ) to infer the grammatical structure of test cases.

Unlike conventional taint analysis techniques, our approach focuses on tracing the propagation of input copies in memory while disregarding the propagation of data that undergoes transformations within the input. We consider the minimal units on the grammar tree to be the input fields that have not undergone any calculation or transformation operations. By capturing the tainted labels associated with these units, we are able to record the hierarchical structure and offset intervals encoded within the tainted labels. Consequently, based on this information, we can reconstruct the grammatical structure tree of the test case.

4.1.1. Lightweight instruction tracking

ScissorFuzz only consider data movement instructions as the propagation pathway for taint. Table 3 presents the distinct handling of program instructions by *ScissorFuzz*. For system call instructions, *ScissorFuzz* generates the initial taint labels based on the length and offset obtained from external inputs. For data movement instructions, taint labels propagate along the movement of data. For data transformation instructions, although these instructions themselves carry information about the data flow, they often represent fine-grained processing logic for the bytes of the same field. Continuing to track the data propagation through these instructions can easily result in the loss of the original field information. Therefore, *ScissorFuzz* discontinues tracking the taint propagation of such instructions.

4.1.2. Hierarchically structured taint labels

Traditional taint analysis typically assigns fixed granularity (e.g., byte, bit) to taint labels and creates them only when the program receives input. In contrast, *ScissorFuzz* creates a new taint label with hierarchical information during each taint propagation. The structure of the label is as follows: *<id, parent, offsetstart, offsetend, stackinfo>*. Here, "id" serves as a unique primary key for each label, "parent" represents the taint label ID from which the data propagation originates, "offsetstart" and "offsetend" correspond to the offset range in the original test cases, and "stackinfo" encompasses the call stack information of the current node, which provides valuable guidance for subsequent target selection and trimming. As shown in the algorithm 1, during taint propagation, based on the starting offset of the source label

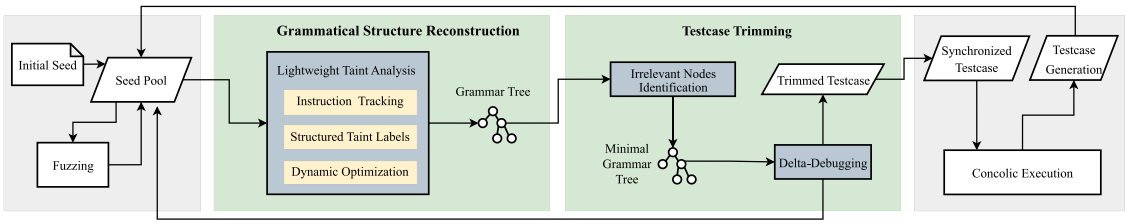


Fig. 6. Overview of *ScissorFuzz*'s architecture.

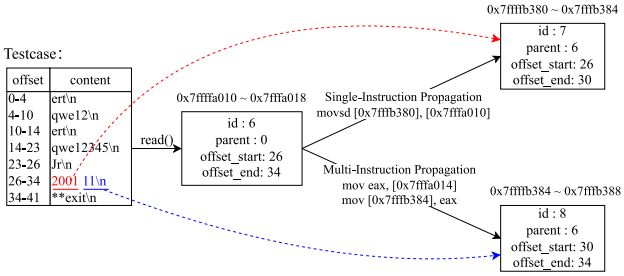


Fig. 7. Propagation process and hierarchy structure of taint labels.

Algorithm 1: Lightweight Taint Propagation

Input: srcAddr, len
Output: <id,parent,offsetstart,offsetend,stackinfo>

```
t ← getTaintTag(srcAddr)
if t then
    firstAddr ← getFirstAddress(t)
    newStart ← t.offsetstart + srcAddr - firstAddr
    newEnd ← newStart + len
    stinfo ← getCurStackInfo()
    return <id, t.id, newStart, newEnd, stinfo>
end
```

and the length being read, we calculate the offset range in the target memory that corresponds to the data in the test case and generate a new taint label accordingly.

We use KPRCA00023 as an example to illustrate how hierarchical taint labels propagate in *ScissorFuzz*. As shown in Fig. 7, the program initially uses a READ system call within a loop to read content from the test case line by line. The input consists of 7 lines of data, and the read content is stored in memory. This process generates 7 sibling nodes with a parent of 0 (default root node), each of which records the offset interval for the corresponding node. Specifically, the 6 line of data corresponds to a taint label with an ID of 6, marking the memory region starting at 0x7fffa010. Since the program performs validation on both the year and month separately, it reads the first 4 bytes and the last 4 bytes from 0x7fffa010 during the validation process. Consequently, taint tracking generates a taint label (ID 7) for the first 4 bytes and another taint label (ID 8) for the last 4 bytes. Both of these taint labels have their parent node pointing to the taint label with an ID of 6. Eventually, *ScissorFuzz* construct a hierarchical taint label tree with a depth of 3. As each taint label records the offset interval of the original input, this taint label tree can be transformed into a grammar tree of the original test case.

4.1.3. Dynamic optimization of taint labels

The source code of a program is the most expressive representation of the grammatical structure of test cases. We observed that relying solely on binary machine instructions to reconstruct the grammatical structure of test cases poses challenges due to the limitations of machine instructions and compiler optimizations. Binary instructions typically handle data blocks that may not precisely correspond to the fields within the grammatical structure of test cases. Each instruction operates on a limited and fixed-length data block. Consequently, relying solely on binary instructions becomes inadequate for accurately reconstructing the grammatical structure of test cases.

Table 4
Machine instructions for fread with different lengths of data read.

Read length	Cache enabled	Machine instructions
5	Yes	syscall(fd, [rsi], 8092) ..... [ins] mov ecx, dword ptr [rsi+rdx*1-0 × 4] [ins] mov esi, dword ptr [rsi] [ins] mov dword ptr [rdi+rdx*1-0 × 4], ecx [ins] mov dword ptr [rdi], esi
65	Yes	syscall(fd, [rsi], 8092) ..... [ins] vmovdqu ymm0, ymmword ptr [rsi] [ins] vmovdqu ymm1, ymmword ptr [rsi+0 × 20] [ins] vmovdqu ymm2, ymmword ptr [rsi+rdx*1-0 × 20] [ins] vmovdqu ymm3, ymmword ptr [rsi+rdx*1-0 × 40] [ins] vmovdqu ymmword ptr [rdi], ymm0 [ins] vmovdqu ymmword ptr [rdi+0 × 20], ymm1 [ins] vmovdqu ymmword ptr [rdi+rdx*1-0 × 20], ymm2 [ins] vmovdqu ymmword ptr [rdi+rdx*1-0 × 40], ymm3
4097	Yes	syscall(fd, [rsi], 8092) ..... [ins] rep movsb byte ptr [rdi], byte ptr [rsi]
n	No	syscall(fd, buf, n)

For instance, fread (buf, size, count, fd) in the libc library, upon its initial invocation, reads a relatively large fixed-length content from a file into a designated memory region for caching purposes. Subsequent calls to the fread function retrieve data of a specified length (count parameter * size parameter) from the cached memory region and store it into the user-specified memory region (buf parameter). Table 4 illustrates the variations in machine instructions used by fread for different specified lengths. It can be observed that due to the fixed data length processed by machine instructions (constrained by various registers' sizes), the movement of a complete data fragment requires the execution of multiple instructions. If taint tracking were to trace each individual data movement instruction, it would not only result in an excessive number of taint labels but also hinder the accurate reconstruction of the underlying grammatical structure.

Therefore, in *ScissorFuzz*, when creating new taint labels, we examine whether there are already existing taint labels at the preceding and succeeding memory addresses. When such labels exist and their offset ranges precisely align with the current offset, *ScissorFuzz* merges these two regions' taint labels. As shown in Fig. 8, when taint analysis labels the memory region at mem+4 as tag2, the adjacent Mem+3 region has already been labeled as tag1, and the corresponding data offsets in the test case are contiguous. At this point, we merge tag1 and tag2 to generate a new label (tag3) that encompasses an entire memory region from mem to mem+8.

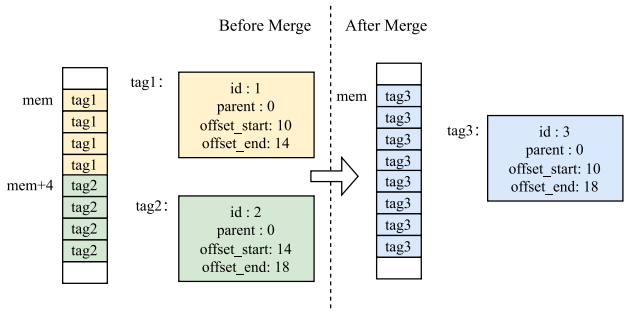


Fig. 8. Dynamic merging of taint labels.

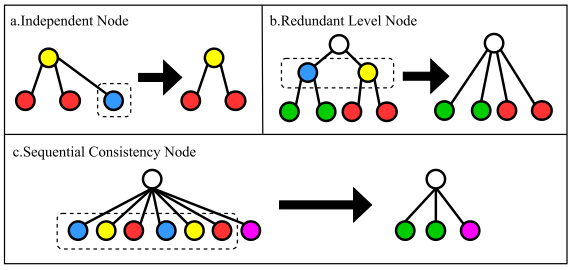


Fig. 9. Three fundamental substructures in the grammar tree.

4.2. Test case trimming

After obtaining the grammar tree of the test case through lightweight taint analysis, it is necessary to determine which nodes in the tree structure can be trimmed. In hybrid fuzzing, the coverage path taken by the program during the execution of a test case is considered a critical factor. For example, AFL treats test cases with new code coverage as interesting. Therefore, we consider changes in code coverage as the basis for determining whether nodes in the tree can be trimmed. We begin by identifying irrelevant nodes on the grammar tree, trimming it down to a minimal grammar tree. Then we utilize a hierarchical delta debugging algorithm to trim nodes within the minimal grammar tree, subsequently trimming the corresponding fields in the test case.

4.2.1. Irrelevant nodes identification

When the number of nodes in the tree is very large, the program is executed multiple times, leading to a significantly slow trimming process. To narrow down the target for trimming attempts, *ScissorFuzz* firstly removes irrelevant nodes and merges redundant nodes to generate a trimmed structure tree. Specifically, we establish specific node structure types and implement a static matching approach to identify and delete nodes that satisfy predefined criteria from the grammar tree of the test case. As shown in Fig. 9, based on the parent–child relationships and the call stack information of nodes in the grammar tree, we define three node types. Different colors in the graph represent different call stack information. And Fig. 10 shows the source code examples of each structure.

Independent node. Independent nodes represent nodes that have no sibling nodes with the same call stack information and do not have any child nodes. This means the data fragments of these nodes are not processed in a loop logic or are executed only once within a loop. Removing data fragments corresponding to such nodes often leads to changes in the program's coverage path. Therefore, it is advisable to delete these nodes in advance.

Redundant level node. Redundant levels indicate nodes that have no sibling nodes with the same call stack information but contain

Independent Nodes	**Redundant Level Node**
1. while (condition): 2. var1 ← read input 3. var2 ← read input	1. buff ← read input 2. while (condition): 3. var1 ← read buff
1. while (condition): 2. var1 ← read buff 3. var2 ← read buff 4. var3 ← read buff	**Sequential Consistency Node**

Fig. 10. Code examples of each structure.

child nodes. For such nodes, after deletion, their child nodes should be preserved and elevated by one level.

Sequential consistency node. Sequential consistency nodes refer to sibling nodes under the same parent node that are arranged in a cross-interleaved manner in time. From the cross-interleaved sequence, obvious loop structures can be identified. It is often corresponds to a situation in the program where multiple different fields are read from the input within a single loop. Since these fields are all within the same loop iteration, deleting only part of the nodes would disrupt the loop structure of the input, leading to alterations in the coverage path. Therefore, when dealing with such substructures, we merge the repeated parts into a single node.

By traversing the nodes in the grammar tree, we ascertain the categorization of each node among the aforementioned three types. Depending on the node's type, we evaluate whether the execution path of the input undergoes any changes before and after its trimming. We identify and mark those nodes that, if trimmed, would affect the program's execution path as irrelevant nodes, subsequently removing them from the grammar tree. In subsequent dynamic trimming, we ensure to avoid trimming these nodes to enhance the efficiency of the delta debugging process.

4.2.2. Hierarchical delta debugging

For the trimmed structure tree obtained through irrelevant nodes identification, *ScissorFuzz* employs a top-down traversal approach to sequentially evaluate each node for deletion. As shown in Algorithm 2, we remove the corresponding offset range of the node and run the program to determine if the coverage path has changed. This strategy ensures that the trimming process gradually transitions from coarse-grained to fine-grained. Once a node is deemed deletable, all its child nodes are trimmed simultaneously. However, when a node is determined to be non-deletable, we traverse its child nodes and perform more granular trimming to ensure the final result is the most streamlined.

5. Implementation

We developed a prototype of the proposed approach, named *ScissorFuzz*. The GitHub link is available at https://github.com/zzyyrr/ScissorFuzz.git. First, we implemented our lightweight taint analysis module using Intel Pin 3.26 (Luk et al., 2005), a dynamic binary instrumentation tool provided by Intel Corporation. The tool demonstrates robust stability and provides extensive instruction set support. It offers support for IA-32 and x86-64 architectures and is compatible with both Linux and Windows operating systems. In *ScissorFuzz*, the taint analysis component consists of more than 650 lines of C++ code. Second, we implemented our test case trimming module on top of the

Algorithm 2: Hierarchical Delta Debugging

```
Input: trimmed grammar tree
Output: trimable node set
Queue={tree}, List={}
while Queue.isNotEmpty() do
    Node ← Queue.front()
    tmptree ← removeNode(finaltree,child)
    if isChangeCov(tmptree) then
        for child ∈ Node.children do
            Queue.push(child)
        end
    else
        List.add(Node)
        tree ← tmptree
    end
end
return List
```

Table 5
Details of the target programs.

Program	Version	Input type	Run command
exiv2	0.25	image	exiv2 @@
imginfo	jasper 2.0.12	image	imginfo -f @@
tiffplit	libtiff 3.9.7	image	tiffplit @@
mp42aac	Bento4 1.5.1–628	video	mp42aac @@ /dev/null
ffmpeg	4.0.1	video	ffmpeg -y -i @@
nm	binutils 5279478	ELF	nm -A -a -l -S -s -D @@

AFL-tmin tool (afl-tmin, 2013) from AFL. In *ScissorFuzz*, we replaced the trimming strategy in AFL-tmin with over 360 lines of C language code.

6. Evaluation

To evaluate the effectiveness of *ScissorFuzz*, we design and conduct comprehensive experiments to answer the following questions:

- **RQ1:** How effective is *ScissorFuzz*'s approach in discovering bugs and achieving better code coverage when fuzzing real-world programs?
- **RQ2:** Can *ScissorFuzz* improve the efficiency of concolic execution?
- **RQ3:** Does the grammar tree generated by *ScissorFuzz* align with the actual grammar structure of the program?
- **RQ4:** Can *ScissorFuzz* efficiently and accurately trim test cases compared to AFL-tmin?

Experimental setup. We run all the following experiments on Linux 4.15.0-142-generic equipped with Intel Xeon E5-2670 (having eight 2.50 GHz cores) and 128 GB RAM. To validate the effectiveness of our proposed grammar-aware test case trimming technique, we selected target programs that accept inputs in specialized formats such as audio, images, and videos. As shown in Table 5, we select 6 real-world target programs provided by UNIFUZZ dataset (Li et al., 2021) that handle different input types. For each selected target program, we conduct comparative experiments for 12 h as done in previous study (Zhao et al., 2019). To evaluate the effectiveness of vulnerability detection, we select QSYM, a state-of-the-art hybrid fuzzing technique, as the baseline technique. To evaluate the trimming efficiency of *ScissorFuzz*, we select AFL-tmin as the baseline technique.

6.1. Performance on code coverage and unique crashes

In order to demonstrate the effectiveness of our test case trimming technique in improving the efficiency of hybrid fuzzing, we conduct comparative experiments between *ScissorFuzz* and QSYM on 6 real-world programs. We use code coverage and the number of unique crashes as evaluation metrics.

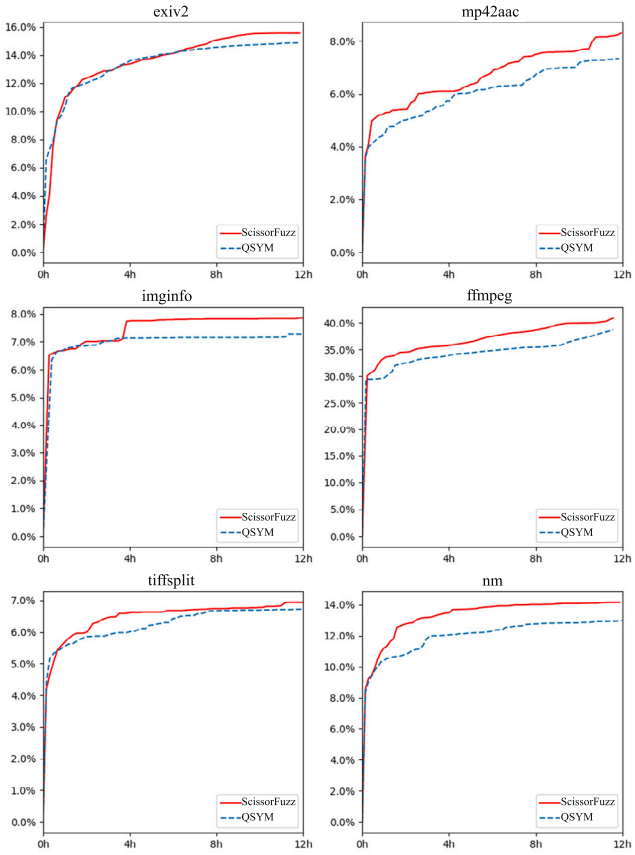


Fig. 11. Average bitmap size for every program.

Code coverage is a critical metric for evaluating the performance of a fuzzing technique (Wang et al., 2019). Basically, the more code a fuzzing technique can cover, the more likely it is to find the hidden bugs. According to previous studies (Lemieux and Sen, 2018; Zhao et al., 2019), we use the bitmap maintained by AFL to measure the code coverage. Specifically, AFL maps each branch transition into an entry of the bitmap via hashing. If a branch transition is explored, the corresponding entry in the bitmap will be filled and the size of the bitmap will increase. As AFL provides real-time bitmap sizes, we can evaluate the increasing code coverage over time during the fuzzing process. First, we present the average bitmap size for every program. As shown in Fig. 11, each graph represents a program, with the abscissa representing the time and the ordinate representing the size of the average bitmap size. It can be observed that *ScissorFuzz* outperforms QSYM on all the programs. In particular, the code coverage of QSYM on 4 out of 6 programs (exiv2, imginfo, tiffsplit, and nm) show a flat trend after 4 h. For example, In mp42aac, *ScissorFuzz* still exhibits an increasing trend in coverage after 12 h of testing, while the variation curve for QSYM remains stable. In the case of ffmpeg, *ScissorFuzz* demonstrates a higher growth rate and consistently maintains a leading coverage.

Table 6 presents the code coverage statistics for each program, indicating that *ScissorFuzz* shows a minimum coverage improvement of 3.27% (tiffplit) and a maximum improvement of 15.7% (nm) compared to QSYM, with an average improvement of 8.2%.

Next, we collect the number of unique crashes discovered in 12 h, as shown in Table 7. It can be observed that only in the case of the tiffsplit program, *ScissorFuzz* triggers fewer unique crashes compared to

Table 6
Final code coverage after 12 h for each program.

Program	QSYM	*ScissorFuzz*	Increase percentage
exiv2	14.89%	**15.55%**	4.43%
imginfo	7.28%	**7.85%**	7.83%
tiffplit	6.72%	**6.94%**	3.27%
mp42aac	7.40%	**8.31%**	12.30%
ffmpeg	38.80%	**40.99%**	5.64%
nm	12.29%	**14.22%**	15.70%

Table 7
Number of discovered unique crashes.

Program	QSYM	*ScissorFuzz*
exiv2	1	**63**
imginfo	0	**2**
tiffplit	**250**	239
mp42aac	123	**185**
ffmpeg	0	0
nm	0	0

Table 8
Code coverage and unique crashes of ScissorFuzz and DigFuzz.

Program	Code Coverage		Unique Crashes	
	DigFuzz	*ScissorFuzz*	DigFuzz	*ScissorFuzz*
exiv2	14.41%	15.55%	48	63
imginfo	7.68%	7.85%	0	2
tiffplit	6.62%	6.94%	186	239
mp42aac	7.88%	8.31%	121	185
ffmpeg	39.20%	40.99%	0	0
nm	13.33%	14.22%	0	0

QSYM. However, for the other programs, *ScissorFuzz* shows significant improvements. Particularly in the case of exiv2, QSYM triggers only 1 unique crash, while *ScissorFuzz* triggers 63 unique crashes.

We also compared the performance of *ScissorFuzz* with the state-of-the-art hybrid fuzzing technique DigFuzz. We evaluated both techniques on the same set of programs, comparing their code coverage and the number of unique crashes detected. As depicted in Table 8, *ScissorFuzz* exhibits higher code coverage than DigFuzz across all programs, and it is able to discover more unique crashes.

By comparing the code coverage and the number of unique crashes, it is evident that through efficient trimming of test cases, *ScissorFuzz* greatly improves the efficiency of hybrid fuzzing in handling highly structured data programs.

6.2. Performance of concolic execution engine

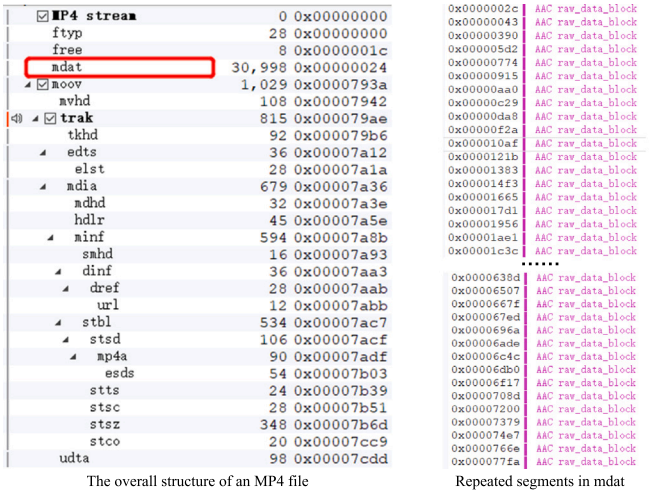
To investigate the impact of test case trimming on concolic execution, we compared the number of test cases generated by *ScissorFuzz*'s concolic execution engine with QSYM's in 12-hour. The greater the number of test cases means that more paths are discovered by the concolic execution.

The experimental results are presented in Table 9. In Table 9, "Executions" represents the number of times concolic execution is launched in 12 h. "Test cases" indicates the number of test cases generated by concolic execution, and "Average" denotes the average number of test cases generated per execution. The results show that *ScissorFuzz* generates a higher number of effective test cases compared to QSYM. In the case of exiv2, although *ScissorFuzz* performed fewer executions than QSYM, it ultimately produced more test cases. This is because *ScissorFuzz* removes redundant data from test cases, preventing the concolic execution from getting trapped in loops and generating repetitive invalid test cases.

These results indicate that redundant data can degrade the performance of concolic execution. By trimming test cases, the efficiency of generating test cases through concolic execution is effectively improved, validating the effectiveness of our approach.

Table 9
The solving time and number of generated test cases of concolic execution.

Program	QSYM			*ScissorFuzz*		
	Executions	Test cases	Average	Executions	Test cases	Average
exiv2	718	1640	2.28	619	**1737**	**2.81**
imginfo	844	856	1.01	975	**1032**	**1.06**
tiffplit	467	**769**	**1.65**	466	763	1.64
mp42aac	776	1946	2.51	945	**2436**	**2.58**
ffmpeg	457	1	0.00	411	**315**	**0.77**
nm	699	1667	2.38	697	**1688**	**2.42**



The overall structure of an MP4 file — Repeated segments in mdat

Fig. 12. Real structure of the test case in mp42aac.

6.3. Accuracy of grammar tree construction

The grammar tree of *ScissorFuzz* is the base of test case trimming. The correct restoration of the program's underlying grammar structure is essential for better inference and reconstruction of data fragments within test cases. In this section, we use mp42aac as an example to demonstrate the correctness of the grammatical structure restored by our approach.

Mp42aac is used to extract audio information from MP4 video files and generate AAC format audio files. Fig. 12 shows the analysis results of a test case in mp42aac using the video stream analysis software called Elecard Stream Analysis. From its structure, it can be divided into four data segments: ftyp, free, mdat, and moov. Among them, ftyp represents the file format information, free represents reserved space information, and special attention should be paid to mdat and moov. Mdat contains the media data of the entire video, while moov primarily records metadata about the video file, such as duration and track information. The execution flow of mp42aac involves first reading the metadata in the moov structure and then reading the media data in the mdat based on the acquired information.

We use *ScissorFuzz* to restore the grammatical structure of this test case, as shown in Fig. 13. From the overall tree shape, there are some repeated subtree structures on the left side, while there is a large subtree structure on the right side. By analyzing the offset information of the nodes, it can be determined that the left-side nodes correspond to the mdat data segment, while the right-side nodes correspond to the moov data segment. By comparing the results generated by the analysis tool, it can be verified that *ScissorFuzz* accurately restores the grammatical structure corresponding to the test case. Furthermore, a significant number of the right-side nodes can be directly deleted after irrelevant nodes identification, demonstrating the efficiency of our technique as only a small number of the left-side nodes need to undergo subsequent delta debugging.

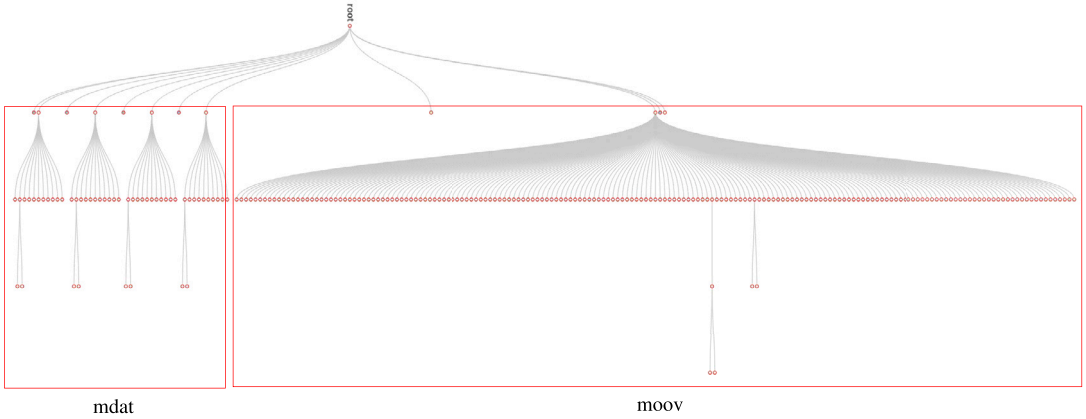


Fig. 13. Grammar tree restored by *ScissorFuzz*.

Table 10
The performance of *ScissorFuzz* on different input structures.

Program	Main structure	*ScissorFuzz*
exiv2	Hoffman table in JPG file	✓
imginfo	Headers in JPG files	✓
tiffsplit	IFD (Image File Directory) in TIF files	✓
mp42aac	List of ACC_DATA_BLOCK in MP4 files	✓
ffmpeg	The "movi" block in the AVI file	✓
nm	List of segments in ELF files list of symbols in symtab segments	✓

Table 11
Results of trimming efficiency.

Program	Test cases size (Byte)	AFL-tmin		*ScissorFuzz*	
		Time (s)	Ratio	Time (s)	Ratio
exiv2	33 726.05	321.24	6.56%	**15.03**	3.95%
imginfo	265 407.75	1167.56	30.37%	**17.48**	32.91%
tiffplit	20 117.45	681.17	16.88%	**3.81**	11.20%
mp42aac	46 567.15	216.72	25.18%	**21.7**	16.30%
ffmpeg	743 061.32	4813.06	4.03%	**25.1**	3.13%
nm	729 659.05	3101.89	0%	**7.85**	0.01%

Apart from mp42aac, we conduct a comprehensive analysis of all the remaining programs in the experiment. We manually examined the processing logic of the programs and the main structures involved in their execution. Subsequently, we compared the grammar trees reconstructed by *ScissorFuzz* with the original ones to validate the restoration of the program's main structure. The verification results are presented in Table 10.

ScissorFuzz uses dynamic taint analyze during program execution to reconstruct input structures. Therefore, the program's execution flow and the resulting reconstructed structures may vary when different run time parameters are used. The programs reconstructed in this section were based on the run time parameters listed in Table 5. Among all the programs, *ScissorFuzz* is capable of restoring its main list structure, demonstrating the effectiveness of our approach.

6.4. Trimming speed and redundancy reduction performance

For test case trimming, the speed and the ratio of data segments that are trimmed are the most important indicators. To evaluate the trimming efficiency of *ScissorFuzz*, we select AFL-tmin as a baseline technique. AFL-tmin is a tool provided by AFL for test case trimming, and both tools aim to maintain the same bitmap coverage before and after trimming while minimizing the size of test cases.

For each program, we select the 20 largest test cases from the test case generated in the experiments and perform trimming using AFL-tmin and *ScissorFuzz* separately. We collect the average time and percentage of trimming for each trimming process. Due to the iterative trimming strategy of AFL-tmin, the trimming process for some test cases exceeded 12 h. To address this, we limit the trimming time to 12 h and calculated the trimming percentage based on the size already trimmed during the execution. The results are shown in Table 11.

It can be observed that compared to AFL-tmin, *ScissorFuzz* has a significant advantage in trimming time, with an average trimming time of fewer than 20 s, much lower than AFL-tmin. However, in terms of trimming percentage, AFL-tmin slightly outperforms *ScissorFuzz*. This is because *ScissorFuzz* does not trim the input portions of the program that are not read, whereas AFL-tmin's arbitrary trimming may remove these parts of the data. This is also a significant reason for the low efficiency of AFL-tmin.

Furthermore, AFL-tmin exhibits variations in trimming time for different programs. This is attributed to AFL-tmin's lack of awareness of the structural information of test cases. During the trimming process, AFL-tmin blindly selects and removes fragments based on a fixed strategy. Once a fragment is successfully removed, the generation of a new test case requires restarting the iteration. Such iterative operations result in significant differences in the upper and lower bounds of the program's execution time, making the trimming speed unstable. In contrast, *ScissorFuzz* trims fragments based on the structure of test cases, with a relatively fixed number of trimming iterations. Compared to different programs and test cases, its trimming speed remains stable, leading to better overall usability than AFL-tmin.

7. Conclusion

In this paper, we observe that the redundant fields in test cases hamper the efficiency of concolic execution. Building upon an in-depth study of the principles of fuzzing and concolic execution, we propose a test case trimming approach to eliminate redundant fields from test cases, and implement a prototype *ScissorFuzz*. Experimental results show that, *ScissorFuzz* outperforms QSYM in code coverage, crash detection, and the efficiency of concolic execution. Experimental results confirm the correctness and effectiveness of our proposed approach.

CRediT authorship contribution statement

Yiru Zhao: Conceptualization, Writing – original draft. **Long Gao:** Formal analysis, Visualization. **Qihan Wan:** Validation, Software, Investigation. **Lei Zhao:** Resources, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

afl-tmin, 2013. Afl-tmin. http://www.tin.org/bin/man.cgi?section=1&topic=afl-tmin.

Alsaedi, A., Alhuzali, A., Bamasag, O., 2022. Effective and scalable black-box fuzzing approach for modern web applications. J. King Saud Univ. Comput. Inf. Sci. 34 (10, Part B), 10068–10078. http://dx.doi.org/10.1016/j.jksuci.2022.10.006, URL: https://www.sciencedirect.com/science/article/pii/S1319157822003573.

Aminu Muazu, A., Sobri Hashim, A., Sarlan, A., Abdullahi, M., 2023. SCIPOG: Seeding and constraint support in IPOG strategy for combinatorial t-way testing to generate optimum test cases. J. King Saud Univ. Comput. Inf. Sci. 35 (1), 185–201. http://dx.doi.org/10.1016/j.jksuci.2022.11.010, URL: https://www.sciencedirect.com/science/article/pii/S1319157822004086.

Barisal, S.K., Chauhan, S.P.S., Dutta, A., Godboley, S., Sahoo, B., Mohapatra, D.P., 2022. BOOMPizer: Minimization and prioritization of CONCOLIC based boosted MC/DC test cases. J. King Saud Univ. Comput. Inf. Sci. 34 (10 Part B), 9757–9776. http://dx.doi.org/10.1016/J.JKSUCI.2021.12.007.

Chen, Y., Ahmadi, M., Wang, B., Lu, L., et al., 2020. {MeuZZ}: Smart seed scheduling for hybrid fuzzing. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses. RAID 2020, pp. 77–92.

Chen, P., Chen, H., 2018. Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 711–725.

Chen, J., Han, W., Yin, M., Zeng, H., Song, C., Lee, B., Yin, H., Shin, I., 2022. {SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis. In: 31st USENIX Security Symposium. USENIX Security 22, pp. 2531–2548.

Chipounov, V., Kuznetsov, V., Candea, G., 2011. S2E: A platform for in-vivo multi-path analysis of software systems. Acm Sigplan Not. 46 (3), 265–278.

Cho, M., Kim, S., Kwon, T., 2019. Intriguer: Field-level constraint solving for hybrid fuzzing. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 515–530.

DARPA, 2017. Cyber grand challenge challenge repository. http://www.lungetech.com/cgc-corpus/.

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.

honggfuzz, 2010. Honggfuzz. https://github.com/google/honggfuzz.

Hosseini, S.M.J., Arasteh, B., Isazadeh, A., Mohsenzadeh, M., Mirzarezaee, M., 2021. An error-propagation aware method to reduce the software mutation cost using genetic algorithm. Data Technol. Appl. 55 (1), 118–148. http://dx.doi.org/10.1108/DTA-03-2020-0073.

Huang, H., Yao, P., Wu, R., Shi, Q., Zhang, C., 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1613–1627.

Lemieux, C., Sen, K., 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 475–485.

Li, Y., Chen, B., Chandramohan, M., Lin, S.-W., Liu, Y., Tiu, A., 2017. Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 627–637.

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.

Lin, P.h., Hong, Z., Li, Y.h., Wu, L.f., 2021. A priority based path searching method for improving hybrid fuzzing. Comput. Secur. 105, 102242.

Liow, L.H., Van Valen, L., Stenseth, N.C., 2011. Red queen: from populations to taxa and communities. Trends Ecol. Evol. 26 (7), 349–358.

Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M., 2005. Pin: building customized program analysis tools with dynamic instrumentation. In: Sarkar, V., Hall, M.W. (Eds.), Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation. Chicago, IL, USA, June 12–15, 2005, ACM, pp. 190–200. http://dx.doi.org/10.1145/1065010.1065034.

Majumdar, R., Sen, K., 2007. Hybrid concolic testing. In: 29th International Conference on Software Engineering. (ICSE 2007), Minneapolis, MN, USA, May 20–26, 2007, IEEE Computer Society, pp. 416–426. http://dx.doi.org/10.1109/ICSE.2007.41.

Mi, X., Rawat, S., Giuffrida, C., Bos, H., 2021. LeanSym: Efficient hybrid fuzzing through conservative constraint debloating. In: Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 62–77.

Misherghi, G., Su, Z., 2006. HDD: hierarchical delta debugging. In: Proceedings of the 28th International Conference on Software Engineering. pp. 142–151.

Poeplau, S., Francillon, A., 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In: 29th USENIX Security Symposium. USENIX Security 20, pp. 181–198.

Shomali, N., Arasteh, B., 2020. Mutation reduction in software mutation testing using firefly optimization algorithm. Data Technol. Appl. 54 (4), 461–480. http://dx.doi.org/10.1108/DTA-08-2019-0140.

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, Vol. 16, No. 2016. pp. 1–16.

Wang, J., Duan, Y., Song, W., Yin, H., Song, C., 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses. RAID 2019, Chaoyang District, Beijing, China, September 23–25, 2019, USENIX Association, pp. 1–15, URL: https://www.usenix.org/conference/raid2019/presentation/wang.

Wang, T., Wei, T., Gu, G., Zou, W., 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy. IEEE, pp. 497–512.

Xie, Z., Cui, Z., Zhang, J., Liu, X., Zheng, L., 2020. Csefuzz: fuzz testing based on symbolic execution. IEEE Access 8, 187564–187574.

Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T., 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium. USENIX Security 18, pp. 745–761.

Zalewski, M., 2015. American fuzzy lop. http://lcamtuf.coredump.cx/afl.

Zhao, L., Duan, Y., Yin, H., Xuan, J., 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing.. In: NDSS. pp. 1955–1973.