Boosting Parallel Fuzzing With Boundary-Targeted Task Allocation and Exploration

Hong Liang, Yijia Guo, Haotian Wu, Yifan Xia, Yi Xiang, Xiantao Jin, Hao Peng, Xuhong Zhang, and Shouling Ji, *Member, IEEE*

***Abstract*—As software systems grow in complexity, scale, and update frequency, parallel fuzzing has become essential for mitigating the efficiency limitations of traditional fuzzing. Effective task allocation is vital for maximizing parallel fuzzing efficiency and has garnered significant attention. However, current strategies often overlook critical code areas, treating all regions uniformly, which results in suboptimal exploration. To address the limitations of current approaches, we present FLEXFUZZ, a novel parallel fuzzing system. First, we identify boundary basic blocks that connect covered and uncovered areas, dynamically adapting them as fuzzing progresses. Second, we introduce a boundary-sensitive task allocation scheme that assigns fuzzing tasks based on the identified boundary basic blocks and their potential for exploration. Finally, to ensure focused exploration, we implement a multi-target, distance-guided approach that directs each instance to concentrate on its relevant task area. We have implemented a prototype of FLEXFUZZ and comprehensively evaluated it against the state-of-the-art parallel fuzzing systems. Across standard benchmarks, FLEXFUZZ surpasses other parallel tools: it increases coverage by 18.17% over the next best tool (PAFL), and identifies 33.75% more vulnerabilities than the next best tool (AFL++).**

***Index Terms*—Fuzzing, parallelism, dynamic analysis.**

TABLE I
COMPARISON OF PARALLEL FUZZING SYSTEMS

Parallel Fuzzer	Sgl	Div	Seed	Fix	Dyn	Str	Crt
AFL++ [7]	✓						
EnFuzz [8]		✓					
Cupid [9]		✓					
autofz [10]		✓					
P-Fuzz [11]	✓		✓		✓		
UltraFuzz [12]	✓		✓		✓		
PAFL [13]	✓			✓			
μFUZZ [14]	✓		✓	✓	✓		
AFL-EDGE [15]	✓				✓		
AFLTeam [16]	✓				✓	✓	
FLEXFUZZ	✓				✓	✓	✓

"Sgl" and "Div" denote the use of single or multiple fuzzer types.
"Seed" ensures each seed is assigned to only one instance.
"Fix" and "Dyn" indicate static or dynamic resolution of task conflicts.
"Str" accounts for program structure.
"Crt" prioritizes critical areas over uniform program treatment.

I. INTRODUCTION

FUZZING is widely adopted for identifying software vulnerabilities by generating diverse inputs [1], [2], [3], [4]. However, the increasing complexity and frequent updates of modern software pose significant challenges to fuzzing, requiring efficient and scalable vulnerability discovery. To address these, parallel fuzzing with multicore processors and high-performance computing provides a promising solution for comprehensive vulnerability detection [5], [6].

Parallel fuzzing, exemplified by AFL++ [7], runs multiple fuzzer instances simultaneously. However, simply launching a single fuzzer type often leads to overlapping code exploration due to identical workflows. Therefore, recent works propose various techniques to address this issue, as shown in Table I. Ensemble fuzzing ("Div" in Table I) combines different fuzzer types but may still produce similar behavior due to shared guidance strategies, as demonstrated by Liang et al. [13]. Dynamic seed distribution ("Seed" in Table I) further reduces overlap by assigning each seed exclusively to one specific instance. Nonetheless, Gu et al. [17] highlight that it overlooks the relationships and differences between code areas; the distributed seeds triggering similar code tend to produce duplicate coverage on the same paths.

In contrast to seed-level distribution, which is random and offers no logical coverage guarantee, **task allocation** approaches have been proposed to assign distinct code regions to each instance according to well-defined rules. A fundamental principle of task allocation is that tasks should remain mutually exclusive, as demonstrated by PAFL [13], μFUZZ [14], and AFL-EDGE [15] in Table I. However, such partition-

Received 8 April 2025; revised 14 September 2025; accepted 7 October 2025. Date of publication 17 October 2025; date of current version 6 November 2025. This work was supported in part by NSFC under Grant U244120033, Grant U24A20336, Grant 62172243, Grant 62402425, and Grant 62402418; in part by the Key Project of the National Natural Science Foundation of China under Grant 62536007; in part by China Postdoctoral Science Foundation under Grant 2024M762829; in part by Zhejiang Provincial Natural Science Foundation under Grant LD24F020002; in part by the "Pioneer and Leading Goose" Research and Development Program of Zhejiang under Grant 2025C01082, Grant 2025C02033, and Grant 2025C02263; and in part by Zhejiang Provincial Priority-Funded Postdoctoral Research Project under Grant ZJ2024001. The associate editor coordinating the review of this article and approving it for publication was Dr. Yue Zhang. *(Corresponding author: Shouling Ji.)*

Hong Liang and Shouling Ji are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China (e-mail: hongliang@zju.edu.cn; sji@zju.edu.cn).

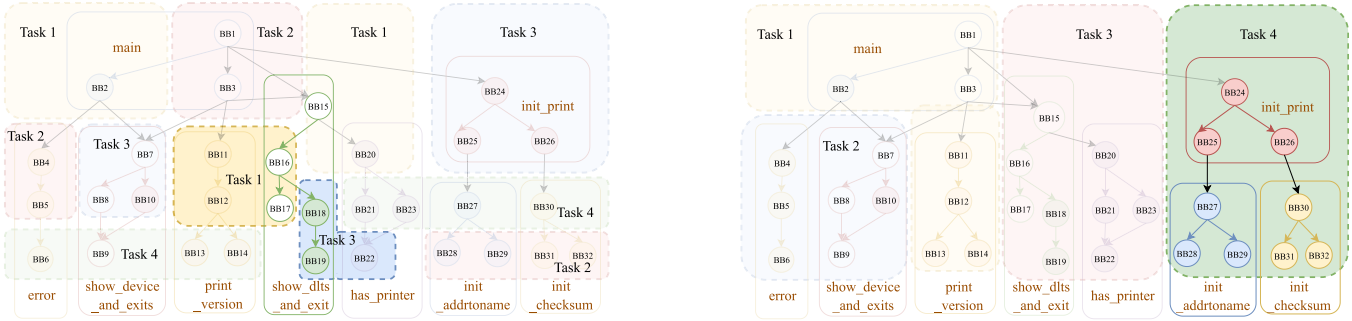
Yijia Guo and Hao Peng are with the School of Computer Science and Technology, Zhejiang Normal University, Jinhua, Zhejiang 321004, China (e-mail: de3mond@zjnu.edu.cn; hpeng@zjnu.edu.cn).

Haotian Wu and Xuhong Zhang are with the School of Software Technology, Zhejiang University, Ningbo, Zhejiang 315048, China (e-mail: dalewu@zju.edu.cn; zhangxuhong@zju.edu.cn).

Yifan Xia and Yi Xiang are with the College of Control Science and Engineering, Zhejiang University, Hangzhou, Zhejiang 310027, China (e-mail: yfxia@zju.edu.cn; xiangyi0406@zju.edu.cn).

Xiantao Jin is with China Electronic Product Reliability and Environmental Testing Research Institute, Guangzhou, Guangdong 511370, China (e-mail: jinxiantao@ceprei.com).

Digital Object Identifier 10.1109/TIFS.2025.3622956



(a) Mutually-exclusive Task Allocation

(b) Structure-aware Task Allocation

Fig. 1. Examples of task allocation on a simplified control flow graph of tcpdump. Functions are shown as rounded rectangles, and basic blocks are shown as circles. Hollow circles represent already-covered blocks, while solid circles represent uncovered ones. (a) Mutually exclusive allocation only separates work across workers with non-overlapping regions. (b) Structure-aware allocation performs a global partition guided by function and CFG structure.

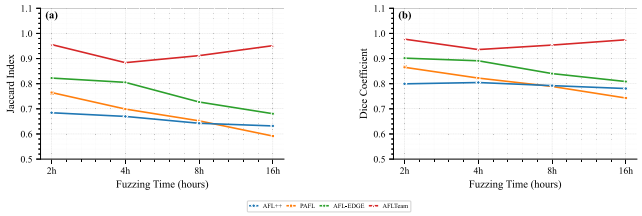


Fig. 2. Task overlap on tcpdump measured by average Jaccard Index and Dice Coefficient (higher = more overlap) across parallel fuzzing strategies at 2, 4, 8, and 16 hours. The two panels report the two metrics, respectively.

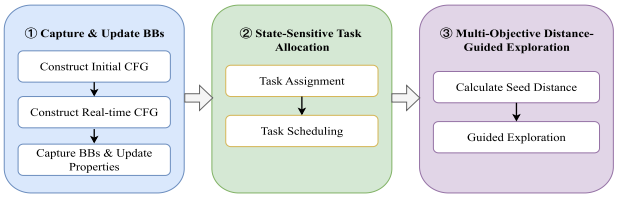


Fig. 3. High-Level Pipeline of FLEXFUZZ.

ing often results in **weak control-flow locality within tasks**, meaning that the basic blocks assigned to the same instance may not be strongly connected in execution paths. Optimized approaches, such as AFLTeam [16], feature function-level structural partitioning and dynamic reprioritization. However, they **overlook the fact that not all code areas require equal attention during fuzzing**, leaving some subtasks largely unexplored. Consequently, fuzzing may become biased toward easily reachable areas, while *hard-to-reach regions lacking initial seed coverage* remain unexplored, ultimately leading to inefficient utilization of computational resources.

Traditional fuzzing typically begins with primary functions and common paths, gradually progressing to less frequently accessed areas. As fuzzing progresses, small critical areas often have the most significant impact on exploration. Fuzzers like K-Scheduler [18], AFLRUN [19], and FOX [20] highlight "horizon nodes", "critical blocks", or "frontier branches" between covered and uncovered regions, which are more likely to reveal new paths. Inspired by this, we define the **boundary basic blocks** shown by the gray dotted line in Figure 5. These basic blocks offer a higher probability of discovering new paths and are adaptable to different programs. As testing progresses, the boundary evolves, reflecting the current fuzzing exploration state and highlighting high-potential areas for future testing. In parallel fuzzing, a straightforward approach is to partition boundary basic blocks into tasks, with each instance focusing on its own task. However, effectively utilizing these boundary basic blocks remains a complex problem. We show the challenges below.

Challenge 1: How to divide tasks reasonably based on boundary basic blocks? Similar to mutually exclusive strategies, boundary basic blocks can be randomly divided into subsets, each containing a comparable number of nodes and assigned to individual fuzzing instances. However, as previously discussed, this approach may weaken **control-flow locality**, defined here as whether the basic blocks in a task lie on adjacent paths in the CFG. Consequently, one fuzzing instance may be forced to explore disjoint fragments, reducing its effectiveness. The real challenge is to partition boundary basic blocks into tasks while respecting control-flow locality. The partition must also adapt to the evolving exploration state, keeping each worker focused while limiting overlap.

Challenge 2: How to achieve a focused exploration of subtasks in fuzzing instances? An intuitive approach is to treat all boundary basic blocks in a task as targets and use directed fuzzing to guide exploration. However, as exploration progresses and boundary basic blocks evolve, tasks must be updated dynamically, complicating seed distance calculations and scheduling strategies. Furthermore, each task often includes numerous boundary basic blocks that require timely exploration. Focusing on all these nodes simultaneously increases computational complexity and may result in scattered exploration.

To address these challenges, we propose FLEXFUZZ, a boundary-targeted parallel fuzzing system. For **Challenge 1**, we propose a boundary-sensitive task allocation scheme with lightweight task assignment and adaptive scheduling. We begin by filtering out shallow, challenging boundary basic blocks and applying extra focus to their exploration. The remaining boundary basic blocks are then split depth-first, expanding tasks within neighboring control-flow regions to preserve control-flow locality. Then, we utilize adaptive scheduling that adjusts tasks dynamically according to coverage growth

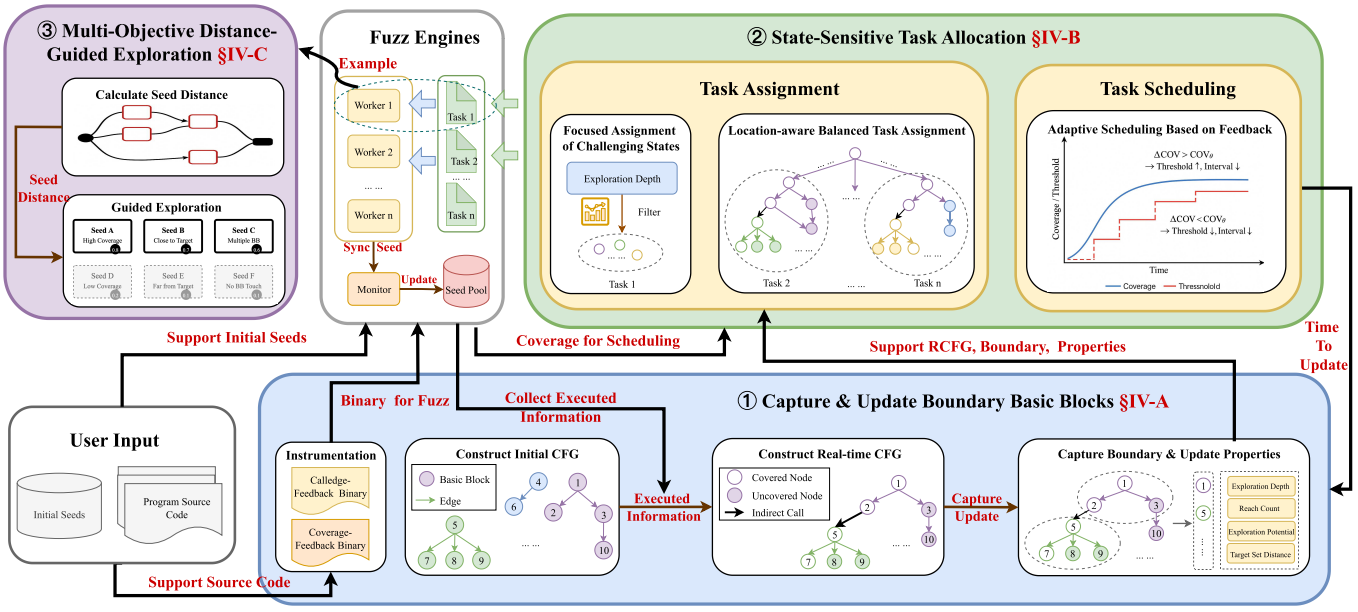


Fig. 4. **Framework of FLEXFUZZ.** The core pipeline has three modules in feedback loops: **Capture and Update Boundary Basic Blocks** maintains the real-time control flow graph and boundary basic block set; **State Sensitive Task Allocation** forms and reschedules tasks; **Multi-Objective Distance-Guided Exploration** advances execution and updates the seed pool. Gray boxes mark contextual components rather than new contributions. *User Input* provides the target program code and initial seeds, and *Fuzz Engines* denote the parallel fuzzers that execute tasks; in our setting, all workers run AFL++. Execution feedback from *Fuzz Engines* flows back to the first and second modules, refreshing the boundary set and triggering rescheduling. After rescheduling, *Fuzz Engines* run the updated tasks by the third module to provide directed guidance and update the seed pool. These outcomes form the next round of feedback.

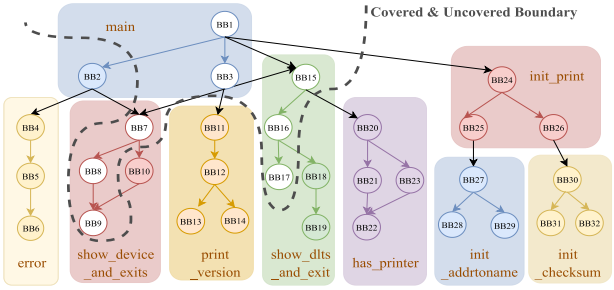


Fig. 5. Simplified real-time control flow graph. Hollow circles denote covered basic blocks, solid circles denote uncovered ones, and the gray dotted line marks the dynamic exploration boundary.

rates. For **Challenge 2**, we recognize that not all boundary basic blocks need targeted exploration, as many can be triggered during normal fuzzing. To tackle this, we introduce a multi-target distance-guided method that balances broadening subtask exploration with targeting complex branches. We evaluate and filter boundary basic blocks within each task, selecting rare ones for directed exploration with a tailored seed selection and energy allocation strategy.

We have implemented our design on top of AFL++ and conducted a thorough evaluation across nine large-scale, widely-used programs, each containing more than 40,000 branches. The results show that FLEXFUZZ increases coverage by 18.17% over PAFL, the next-best tool, and discovers 33.75% more vulnerabilities than AFL++ running in parallel mode. We also perform preliminary experiments to assess each component and analyze the impact of our design. Additionally, we introduce new metrics for analyzing task conflicts and discuss their implications.

In summary, we make the following contributions:

- **Boundary-targeted Parallel Fuzzing Framework:** We account for the critical areas for incremental fuzzing exploration by treating boundary basic blocks as tasks in parallel fuzzing. Building on this foundation, we develop a boundary-sensitive task allocation scheme and a multi-target distance-guided exploration strategy.
- **Implementation of FLEXFUZZ:** Following the design principles, we extend AFL++ to implement FLEXFUZZ. These principles are not specific to AFL++ and can be adapted to other fuzzers. The implementation will be open-sourced to facilitate future research.
- **Comprehensive Evaluation:** We comprehensively compare FLEXFUZZ with state-of-the-art parallel fuzzing systems. The results show an average of approximately 20% improvement in coverage and a 1.34× increase in discovered vulnerabilities over others on our benchmark.

II. BACKGROUND

A. Parallel Fuzzing

Parallel fuzzing enhances testing efficiency by running multiple instances simultaneously. However, designing an effective parallel fuzzer poses several challenges, including maintaining diversity across instances, avoiding the redundant execution of the same seeds, and resolving task conflicts. Various fuzzing systems have adopted different strategies to tackle these challenges, as summarized in Table I, which compares representative parallel fuzzers along several design

dimensions. For instance, the columns "Sgl" and "Div" indicate whether a single fuzzer or multiple types are deployed: traditional systems such as AFL [21] and AFL++ [7] run a single fuzzer per instance, whereas ensemble-based approaches like EnFuzz [8], Cupid [9], and autofz [10] integrate multiple fuzzers to enhance coverage diversity. Similarly, the "Seed" column denotes exclusive per-seed assignment to one engine at a time, following P-Fuzz [11] and UltraFuzz [12], thereby preventing duplicate executions. Task conflict resolution is handled differently: PAFL [13] applies static bitmap partitioning ("Fix"), AFL-EDGE [15] leverages dynamic execution edge information ("Dyn"), and μFUZZ [14] combines static and dynamic allocation to balance workload distribution and feedback utilization. Other systems incorporate structural information, as indicated in the "Str" column. For instance, AFLTeam [16] leverages function-level structure to form macro-tasks from call-graph partitions, then dynamically reallocates effort among those partitions. Despite these advances, existing designs still face practical limitations, including under-prioritizing critical regions and a lack of subtask-focused mechanisms.

B. Directed Fuzzing

Directed fuzzing focuses on specific program locations rather than random exploration. Directed fuzzers such as AFLgo [22], Hawkeye [23], and Beacon [24] prioritize seeds that are closer to target locations or discard inputs that cannot reach them to maintain direction toward the targets. While effective for single-target scenarios, these approaches struggle with multiple target locations. Inaccurate distance calculations can reduce the effectiveness of distance-based scheduling, such as relying on harmonic averages or treating all targets equally, causing these methods to revert to traditional coverage-guided fuzzing. Although recent studies [25], [26], [27] have proposed solutions for multi-target scenarios, these approaches predefine the target set and do not support adding or removing targets during fuzzing. As a result, the challenge of adapting to dynamically updated targets during fuzzing remains unresolved.

III. Motivation

Parallel fuzzing loses efficiency when independent workers revisit the same paths. To make this overlap problem concrete, we examine *tcpdump* and visualize task allocation on a simplified control flow graph in Figure 1. To quantify the repetition rate, we collect each worker's reached-edge set and compute the average pairwise Jaccard Index and Dice Coefficient across workers. Larger values indicate greater overlap, whereas smaller values indicate more effective parallelization. Formal definitions appear in Section VI and online [28]. Figure 2 plots these metrics over time for several parallel fuzzers on *tcpdump*.

Existing approaches split work in two ways. Mutually exclusive allocation assigns disjoint exploration regions to different workers. PAFL and AFL-EDGE follow this idea. Figure 1a shows the consequence on *tcpdump*: contiguous code is fragmented and the five blocks of `show_dlts_and_exit` are split across tasks, so workers do not diverge early and the measured early overlap is high. Structure-aware allocation creates function-level groups and balances effort across them. AFLTeam embodies this design. Figure 1b shows that most trials concentrate on frequently executed functions, while the initialization-heavy Task 4 remains underexplored, keeping overlap high. Measured on *tcpdump*, Figure 2 reveals two consistent phenomena across both metrics: The repetition rate is high, and it does not decline over time. For example, AFL++ starts below PAFL and later rises above it, while AFL-EDGE and AFLTeam remain above AFL++. Overall, longer runs do not reduce overlap across all evaluated parallel fuzzing systems. The root cause is **a mismatch between where tasks are split and where overlap actually accumulates**: when cuts lie deeper than the boundary between covered and uncovered code, many workers still traverse the same covered region before reaching new states; as reachability expands, a fixed or coarse split lags the moving boundary, and overlap plateaus or grows. To link repetition with effectiveness, Table II reports edge coverage for the same runs as Figure 2. The coverage trend mirrors the overlap curves: (i) AFL-EDGE and AFLTeam, which show persistently higher overlap, achieve lower coverage across most time budgets; (ii) AFL++ leads early at 2h and 4h when its overlap is below PAFL, but as its overlap rises later, PAFL overtakes it at 8h and 16h; and (iii) lower overlap aligns with stronger coverage growth. Evidence from both overlap and coverage indicates that duplicated exploration is the primary bottleneck for parallel fuzzing. We should cut tasks at the moving coverage boundary rather than apply global or deeper, coarser partitions.

TABLE II

COVERAGE ON *tcpdump* (EDGES). WE USE THE SAME RUNS AS IN FIGURE 2, AND COVERAGE AND TASK OVERLAP ARE TWO DIMENSIONS OF THE SAME TEST. BOLD DENOTES THE BEST, AND UNDERLINE DENOTES THE SECOND BEST. THE SAME NOTATION APPLIES TO ALL SUBSEQUENT TABLES

Time	AFL++	PAFL	AFL-EDGE	AFLTeam
2h	**19,262.00**	15,956.00	15,971.00	15,089.00
4h	**21,516.00**	19,943.00	19,784.00	16,902.00
8h	24,075.00	**24,872.00**	23,788.00	20,777.00
16h	25,539.00	**26,021.00**	24,710.00	22,907.00

Therefore, we propose a boundary-targeted parallel fuzzing framework. It partitions at the **boundary basic blocks** while maintaining control-flow contiguity. As the boundary moves, it adapts the partitioning and applies directed guidance to keep each task focused. Details are provided in Section IV.

IV. Design

Figure 3 provides a high-level overview of FLEXFUZZ, highlighting the main modules and their interactions. Each module corresponds to a distinct stage; collectively, the modules outline the workflow. The first module, **Capture and Update Boundary Basic Blocks**, constructs both initial and real-time control flow graphs and continuously maintains up-to-date information on boundary basic blocks throughout execution. Next, **State-Sensitive Task Allocation** assigns and schedules tasks from this information so that

each fuzzing instance explores coherent and non-overlapping regions. Finally, **Multi-Objective Distance-Guided Exploration** directs fuzzing instances toward prioritized targets, balancing broad task-level exploration with focused investigation of challenging regions within the same task. While the high-level pipeline is drawn sequentially for readability, the modules interact via feedback loops rather than running in strict sequence at runtime. As execution progresses, FLEXFUZZ updates the boundary basic block set. When the boundary shifts, the scheduler adjusts the update rate based on the current exploration progress and triggers the next task assignment as needed. These feedback loops coordinate parallel instances and focus effort on the assigned task regions. Figure 4 provides a more detailed view of these interactions, and the following subsections describe each module.

A. Capture and Update Boundary Basic Blocks

1) Normative Definition: We introduce definitions about boundary basic blocks and their properties.

Boundary Basic Block: A basic block is considered "covered" if fuzzing has explored at least one incoming edge; otherwise, it is "uncovered". Two basic blocks connected by a control-flow edge are referred to as a predecessor and its successor. If a covered basic block has uncovered successors, it becomes a boundary basic block. It can be formally defined as follows. Let B denote a basic block and $\text{succ}(B)$ indicate the set of successor basic blocks of B. BB_{C} denotes the set of covered basic blocks, while BB_{UC} denotes the set of uncovered basic blocks. The set of boundary basic blocks, denoted as BB_{BD} is defined as follows:

$$BB_{BD} = \{B \in BB_C \mid \exists B' \in \text{succ}(B) : B' \in BB_{\text{UC}}\} \quad (1)$$

B' is a successor of B. Thus, BB_{BD} connects the covered BB_{C} and uncovered BB_{UC} regions. Based on BB_{BD}, a real-time control flow graph is built to track dynamic control-flow transitions between basic blocks.

Properties: In addition to defining boundary basic blocks, we identify related properties for task allocation and exploration, including exploration depth, reach count, exploration potential, and distance. The first three are discussed below, while distance is elaborated in Subsection IV-C.

Exploration Depth: For a boundary basic block, exploration depth refers to the shortest distance from the entry block of the "main" function to the boundary basic block in the real-time control flow graph. As fuzzing advances, the exploration deepens, causing boundary basic blocks to shift and potentially increase depth. Therefore, the boundary basic blocks dynamically evolve, and the exploration depth closely tied to them experiences similar patterns of change. However, if some boundary basic blocks remain at shallow depths, it suggests that the conditions required for flipping branches are intricate and challenging to resolve.

Reach Count: The difficulty of exploring a boundary basic block correlates with its reach count, which measures how many seeds in the seed pool can trigger it. A higher reach count indicates easier access and better exploration, while a lower count suggests the need for further analysis.

Exploration Potential: A boundary basic block is a covered block that has at least one uncovered outgoing edge. Its exploration potential is the number of successors that have not yet been executed. In practice, we look at the block's outgoing edges and count the successors that remain unseen. This value reflects the number of immediate new paths that can open from the block and is larger for a multiway control statement, such as a `switch` statement. The definition is local and efficient as it inspects only one hop.

2) Capture and Update Processes: Accurate control flow information is crucial for updating boundary basic blocks and their properties. However, indirect calls introduce uncertainty in static analysis, as their targets depend on the program's dynamic states. To address this, we have implemented an LLVM instrumentation pass that records runtime call edges, supplementing missing data, referred to as the "Calledge-Feedback Binary" in Figure 4. Additionally, we maintain a seed pool that stores interesting seeds from parallel fuzzing to aid updates.

As shown in Figure 4, FLEXFUZZ instruments the source code to generate two binaries. The coverage-feedback binary determines whether specific seeds should be retained. In contrast, the calledge-feedback binary updates boundary basic blocks and their properties whenever appending new seeds in the seed pool. FLEXFUZZ then builds a real-time control flow graph by retaining executed basic blocks and edges while adding connections from indirect calls. As parallel fuzzing progresses, the seed pool is continuously updated, prompting a corresponding refresh of the real-time control flow graph. Since the capture and update process depends on program scale and significantly impacts FLEXFUZZ's efficiency, we analyze its overhead online [28].

3) Guiding Example: Figure 5 shows a simplified real-time control flow graph with the gray dotted line marking the exploration boundary. As fuzzing progresses, the boundary moves to reflect the evolving exploration scope. Specific boundary basic blocks (BB1, BB3, BB7, BB15, BB16) can be identified from the graph. As fuzzing advances, the boundary basic blocks and their properties are updated accordingly.

B. Boundary-Sensitive Task Allocation Scheme

Using data on boundary basic blocks and their properties, we propose a boundary-sensitive task allocation scheme that includes task assignment and scheduling. In **task assignment**, we first identify shallow boundary basic blocks, referred to as challenging nodes, and designate them as standalone tasks (Task 1) handled by a dedicated worker, as illustrated in Figure 4 (Focused Task Assignment for Challenging Nodes). This design ensures that such nodes are not mixed with others, since their complex path constraints often hinder fuzzing progress and reduce task focus. For the remaining boundary basic blocks, we adopt a location-aware balanced assignment, in which closely related blocks in the real-time control flow graph are grouped into distinct tasks (Tasks 2, . . ., n). This step, shown in Figure 4 (Location-aware Balanced Task Assignment), enables each task to be executed by a fuzzing engine for focused exploration. In **task scheduling**, we apply adaptive

scheduling based on runtime feedback, enabling flexibility and responsiveness to evolving exploration status.

1) Focused Task Assignment for Challenging Nodes: As previously discussed, shallow boundary basic blocks are treated as challenging nodes due to their weak execution correlations with others and hard-to-satisfy constraints. Assigning them individually prevents them from being overshadowed by easier targets and enables more effective fuzzing.

We focus on boundary basic blocks with shallow exploration depth defined in Subsection IV-A.1 as challenging. Employing the z-score (Z), we detect and exclude shallow outlier boundary basic blocks. The z-score for a data point X with mean μ and standard deviation σ is calculated as:

$$Z = \frac{X - \mu}{\sigma} \tag{2}$$

It measures how far a data point deviates from the mean in standard deviations. FLEXFUZZ calculates Z for each boundary basic block based on exploration depth, with μ as the average depth of all boundary basic blocks. By setting a threshold ω, FLEXFUZZ filters the boundary basic blocks with $Z < -\omega$, identifying shallow outliers with depths substantially below the average. These selected challenging boundary basic blocks are grouped as a standby task assigned to a dedicated worker for intensive exploration. While alternatives for focused exploration exist, this approach is straightforward and practical.

2) Location-Aware Balanced Task Assignment: Beyond filtering challenging ones for focused exploration, common boundary basic blocks with closely related execution relationships should be grouped into the same task. Furthermore, we consider the exploration potential, as defined in Subsection IV-A.1 to maintain workload balance in task assignment.

We designed a dynamic, specialized task assignment algorithm rather than traditional static, global methods. This algorithm utilizes the real-time control flow graph (RCFG) and a customized Depth-First Search (DFS) technique. The DFS explores nodes by delving deeply into the graph before backtracking, mirroring the sequential execution of program instructions along a single path. Consequently, all possible execution paths branching from a given point are thoroughly analyzed. By adhering to the structure of these execution paths, the DFS identifies nodes directly connected to the current node, preserving the consistency of execution relationships. During DFS traversal of the RCFG, FLEXFUZZ visits all boundary basic blocks and classifies them according to their locations and exploration potential. If the exploration potential of the currently visited boundary nodes exceeds a precalculated threshold, FLEXFUZZ groups them into a single task for thorough exploration and proceeds with the traversal to generate subsequent tasks.

To illustrate the algorithm's functionality, we provide its pseudo-code in Algorithm 1. The algorithm initializes key variables, such as the visited node list (VS), and calculates the exploration potential threshold θ. This threshold normalizes the overall exploration potential across tasks and serves as a reference for balancing workload. It then performs a Depth-First Search (DFS) traversal of the real-time control flow graph (RG), adding encountered common boundary nodes (BB_{CB}) to the current task. When the currently processed node (B_{now}) is identified as a common boundary basic block, the current exploration potential (EP_{now}) is updated. If EP_{now} meets or exceeds θ (lines 20-24), the *CollectNodes* function abstracts the grouping of visited common boundary basic blocks into a task. The task is added to the task list, and EP_{now} is reset. This process repeats until all nodes (n) in RG are visited.

Our partitioning algorithm is lightweight and designed for large-scale real-world programs. As shown in Table III, the number of branches (edges) ranges from over 30,000 to

Algorithm 1 Task Assignment with RCFG

1: **Input:** $RG \leftarrow$ Real-Time CFG, $BB_{CB} \leftarrow$ Common Boundary Nodes, $EP_{CB} \leftarrow$ Exploration Potential for BB_{CB}, $GP \leftarrow$ Number of Groups, $B_m \leftarrow$ Main Entry Node
2: **Output:** $tasks \leftarrow$ List of Tasks
3: /* Initialize and compute threshold */
4: $VS \leftarrow []$ ▷Visited nodes
5: $\theta \leftarrow \frac{\sum EP_{CB}[B] \text{for all } B \in BB_{CB}}{GP}$
6: /* Define function to group nodes */
7: **function** DFS($RG, B_{now}, BB_{CB}, EP_{CB}, \theta, VS, EP_{now}$)
8: **if** $B_{now} \in VS$ **then**
9: **return** 0
10: **end if**
11: VS.append(B_{now})
12: **for** each n in RG.succ(B_{now}) **do**
13: **if** $n \notin VS$ **then**
14: EP_{now} += DFS($RG, n, BB_{CB}, EP_{CB}, \theta, VS, EP_{now}$)
15: **end if**
16: **end for**
17: **if** $B_{now} \in BB_{CB}$ **then**
18: EP_{now} += $EP_{CB}[B_{now}]$
19: **end if**
20: **if** $EP_{now} \geq \theta$ **then**
21: $task \leftarrow$ CollectNodes(VS, B_{now}, BB_{CB})
22: $tasks$.append($task$)
23: $EP_{now} \leftarrow 0$
24: **end if**
25: **return** EP_{now}
26: **end function**
27: **Call:** DFS($RG, B_m, BB_{CB}, EP_{CB}, \theta, VS, 0$)
28: **Output:** $tasks$

TABLE III
TARGET PROGRAMS EVALUATED IN THE EXPERIMENTS

Binary	Project	Version	Type	Block	Branch
tcpdump	Tcpdump	4.8.1	Network	23,039	40,046
nm-new	Binutils	5279478	Binary	33,381	55,526
pdftotext	Xpdf	4.00	PDF	40,371	62,129
sqlite3	SQLite	3.8.9	Database	40,028	64,798
objdump	Binutils	2.28	Binary	47,218	77,470
exiv2	Exiv2	0.26	Image	70,366	105,279
vim	Vim	9.0.1340	Text	172,669	292,118
magick	ImageMagick	7.1.1-0	Image	179,856	300,657
ffmpeg	FFmpeg	4.0.1	Video	347,923	556,673

500,000, making complex partitioning algorithms impractical due to their high time and space overhead.

3) Adaptive Scheduling Based on Feedback: An inappropriate task update interval can negatively impact parallel fuzzing performance. Frequent updates for every coverage change introduce excessive overhead, while fixed intervals fail to adapt to diminishing coverage growth. To resolve this, FLEXFUZZ employs a dynamic scheduling approach based on global coverage growth trends, providing a more comprehensive view of progress than individual sub-engine data.

Inspired by TCP congestion control [29] and the unpredictability of fuzzing, we propose an exponential backoff mechanism for interval adjustments. Initially, update intervals are short but gradually increase as division rounds progress. We set minimum and maximum limits to prevent excessively short or long task allocation intervals. The minimum interval mitigates frequent updates caused by rapid coverage growth, while the maximum interval prevents prolonged delays when coverage stabilizes. The update threshold dynamically adjusts based on coverage changes: if the coverage increment falls below the threshold, the interval extends, and the threshold decreases; if it exceeds the threshold, the interval shortens, and the threshold increases.

Algorithm 2 explains the process clearly. Since coverage growth rates vary across programs, we should set an appropriate initial coverage growth threshold, COV_θ. To determine this threshold, we run parallel fuzzing without task division and set the initial threshold from the coverage increment observed during the first two minimum intervals, ITV_{min} (lines 4–12). Once initialized, task assignment strategies are applied, boundary-targeted exploration begins, and feedback on coverage growth is collected to update the interval ITV_{now} and threshold COV_θ via the *Adjust_Threshold* function.

Algorithm 2 Adaptive Task Scheduling

1: **Input:** $ITV_{min} \leftarrow$ Minimum Interval, $ITV_{max} \leftarrow$ Maximum Interval, $ITV_{scal} \leftarrow$ Interval Scaling Factor, $COV_{\theta scal} \leftarrow$ Coverage Thresh- old Scaling Factor
2: **while** parallel fuzzing is ongoing **do**
3: /*Initial thresholdcalculation.*/
4: **if** $COV_\theta == 0$ **then**
5: /*Common fuzzing forworkers.*/
6: $ITV_{now} \leftarrow ITV_{min} \times 2$
7: parallel_fuzz(ITV_{min})
8: $COV_{prev} \leftarrow$ get_coverage()
9: parallel_fuzz(ITV_{min})
10: $COV_{now} \leftarrow$ get_coverage()
11: $COV_\theta \leftarrow COV_{now} - COV_{prev}$
12: $COV_{prev} \leftarrow COV_{now}$
13: **else**
14: /* Update tasks anddirected fuzzing forITV_{now}. */
15: tasks_fuzzing_until(ITV_{now})
16: $COV_{now} \leftarrow$ get_coverage()
17: $COV_{inc} \leftarrow COV_{now} - COV_{prev}$
18: $COV_{prev} \leftarrow COV_{now}$
19: Adjust_Threshold($COV_{inc}, COV_\theta, ITV_{now}, ITV_{scal}, ITV_{max}$)
20: **end if**
21: **end while**
22: **function** ADJUST_THRESHOLD ($COV_{inc}, COV_\theta, ITV_{now}, ITV_{scal}, ITV_{max}$)
23: **if** $COV_{inc} < COV_\theta$ **then**
24: $ITV_{now} \leftarrow \min(ITV_{now} \div ITV_{scal}, ITV_{max})$
25: $COV_\theta \leftarrow COV_\theta \times (1 - COV_{\theta scal})$
26: **else**
27: $ITV_{now} \leftarrow \max(ITV_{now}/ITV_{scal}, ITV_{min})$
28: $COV_\theta \leftarrow COV_\theta \times (1 + COV_{\theta scal})$
29: **end if**
30: **end function**

C. Multi-Target Distance-Guided Exploration

After task assignment, each task includes multiple boundary basic blocks and undergoes dynamic updates during parallel fuzzing (see Subsection IV-B.3). To keep each engine focused on its specific task, we propose a lightweight distance measurement method based on the real-time control flow graph (RCFG) and a guided exploration strategy using this metric.

1) Target and Distance Definition: Targeting assigned boundary basic blocks for each worker is feasible. However, the priority of targets evolves during fuzzing, and too many unrelated targets can lead directed fuzzers to behave like undirected ones. Considering all boundary nodes as targets can increase computational overhead. Unlike the function-level distance metric based on call graphs [26], we propose a lightweight and fine-grained basic-block-level distance metric based on our RCFG. We refine the target selection strategy by prioritizing underexplored boundary nodes. Specifically, we identify rare boundary basic blocks based on their reach counts, selecting those with lower counts as the target set BB_{TTG}. These basic blocks are triggered less frequently and more likely to reveal new behaviors. Section V details the specific identification procedure of rare boundary basic blocks.

After selecting targets, we introduce distance calculation with three components: basic block distance, target set distance, and seed distance. Unlike traditional directed fuzzing, which calculates distances through static analysis and instrumentation, our approach leverages a real-time control flow graph to enable adaptive distance calculations.

Basic Block Distance: For two basic blocks B_1 and B_2 on RCFG, the distance $d(B_1, B_2)$ is defined as follows:

$$d(B_1, B_2) = \begin{cases} \text{len\_short}(B_1, B_2), & \text{if reach}(B_1, B_2) \\ \infty, & \text{if } (\neg\text{reach}(B_1, B_2)) \end{cases} \quad (3)$$

In the above expression, len_short(BB_1, BB_2) denotes the shortest path length between BB_1 and BB_2, while reach(BB_1, BB_2) indicates whether a path exists between them. This definition measures the distance between basic blocks based on the shortest path or assigns an infinite distance if the blocks are unreachable in the RCFG.

Target Set Distance: If we have a basic block B_1 and a target set for one task BB_{TTG}, the distance from B_1 to BB_{TTG} is defined as:

$$d(B_1, BB_{TTG}) = \min_{B_t \in BB_{TTG}} d(B_1, B_t) \quad (4)$$

$d(B_1, B_t)$ represents the distance from basic block B_1 to target block B_t, as defined by the basic block distance metric. This equation calculates the distance from B_1 to each basic block in the target set and selects the minimum distance as the distance to the target set for one task BB_{TTG}. This approach identifies the nearest target basic block within the target set, avoiding arbitrary selections or reliance on average distances, which may overlook the actual closest basic block.

Seed Distance: For a seed S and a target set for one task BB_{TTG}, where $P(S) = \{B_1, B_2, \ldots, B_m\}$ denotes the set of basic blocks executed by the seed S. The seed distance of S is defined as:

$$d(S, BB_{TTG}) = \min_{B_i \in P(S)} d(B_i, BB_{TTG}) \tag{5}$$

$d(B_i, BB_{TTG})$ represents the shortest distance from B_i to any target basic block in BB_{TTG}. The multi-target seed distance $d(S, BB_{TTG})$ adopts the minimum of these target set distances across all basic blocks in the execution path. This metric is useful for identifying the closest seed to the target set while avoiding the influence of maximum values and the mode. In contrast, existing distance metrics often adopt a one-size-fits-all approach, such as the average control-flow distance to targets [22], [30]. Such metrics can introduce biases in seed selection by prioritizing globally optimal but locally suboptimal seeds. They may lead to less effective outcomes, as they risk selecting seeds farther from the intended target basic blocks.

2) Guided Exploration: Similar to other directed fuzzers [24], [30], [31], [32], which balance exploitation and exploration across diverse targets, our approach prioritizes seeds according to their influence on task boundaries. To achieve this, FLEXFUZZ integrates a selection probability and energy adjustment scheme, leveraging the seed characteristics of AFL++. Specifically, seeds that cover more basic blocks in the task boundary set BB_{TBD} are assigned higher selection probabilities, as they are more likely to uncover new behaviors. Seeds with small distances to target blocks also receive increased selection probabilities, controlled by an adjustment factor w_{dt} defined in the Implementation Section to reflect their proximity and potential to trigger new behaviors. Seeds covering both BB_{TBD} and BB_{TTG} receive an additional selection probability bonus. We also optimize the energy distribution by incorporating distance metrics, and the implementation details are shown in Section V. The guided exploration strategy enhances new behavior discovery by balancing focus on critical targets with broader testing of task boundaries. It facilitates in-depth exploration while maintaining a comprehensive exploration of the entire task space.

V. IMPLEMENTATION

We have implemented FLEXFUZZ with task allocation in Python and task exploration in C/C++ based on AFL++.

A. Data Collection

The coverage-feedback binary tracks real-time covered edges, similar to AFL++, while the calledge-feedback binary records each function call's entry basic block and its predecessors, providing accurate indirect call information. We extract the initial control flow graph through static analysis in LLVM's Link-Time Optimization (LTO) mode to align basic block and edge data. By executing seeds from the seed pool with the corresponding binary, we dynamically collect function call data and execution details for basic blocks and edges. For identifying boundary basic blocks, we have implemented a simple heuristic approach as defined in Subsection IV-A.1. When task allocation conditions are met, FLEXFUZZ will update the boundary basic blocks and their properties.

B. Task Assignment

The parameter ω in Subsection IV-B.1 influences worker workload balance. A high value reduces the depth difference between selected nodes and others. However, it increases the exploration workload for the dedicated worker. Therefore, users should manually adjust ω based on their experimental setup. In our experiments, we set ω to 0.1 for balanced workload distribution. Similarly, users can configure the predefined task group number GP in Subsection IV-B.2 to determine the threshold θ for task assignment. However, we recommend matching it with the number of available fuzzing workers to ensure each task has a corresponding worker.

C. Task Scheduling

Other parameters in Subsection IV-B.3, such as interval lengths and scaling factors, can be tuned by users. We set ITV_{min} to 10 minutes, ITV_{max} to 3 hours, ITV_{scal} to 2, and $COV_{\theta scal}$ to 0.1, balancing responsiveness and stability in task scheduling and coverage evaluation.

D. Target Determination

We select boundary basic blocks with reach counts in the lowest 30% as the target set for task BB_{TTG}. Parameter choices are discussed online [28].

E. Selection Probability Increment

We replace the global coverage map size metric in AFL++ with a metric including the seed's coverage of BB_{TBD} and newly triggered edges, alongside other factors. Seeds triggering both BB_{TBD} and BB_{TTG} receive a 20% higher selection probability due to their enhanced contribution potential. The rationale for this threshold is discussed on the project site [28].

As for seeds with small distances to targets, their adjustment factor, w_{dt}, is calculated as the ratio of the average distance of all such seeds and the current seed's distance. This approach promotes diversity and increases the chances of exploring alternative paths. w_{dt} is defined as:

$$w_{dt} = \frac{\frac{1}{N}\sum_{i=1}^{N} d(S_i, BB_{\mathrm{TTG}})}{d(S, BB_{\mathrm{TTG}})} \tag{6}$$

The average distance of all seeds S_i that do not touch BB_{TBD} is $\frac{1}{N}\sum_{i=1}^{N} d(S_i, BB_{\mathrm{TTG}})$, where N is the number of such seeds. In AFL++, each test case is assigned a probability based on a score incorporating factors like execution time and coverage map size, each with its weight adjuster. The adjustment factor for our distance w_{dt} operates similarly.

F. Energy Distribution Optimization

Seeds that trigger both BB_{TBD} and BB_{TTG} receive an energy boost. Seeds closer to BB_{TBD}, even if they do not trigger BB_{TBD}, have their energy adjusted based on distance. Shorter distances, indicating higher exploration potential, result in greater energy multipliers. This adjustment follows the same range as AFL++, ensuring effective performance in FLEXFUZZ.

VI. EVALUATION

We conduct a comprehensive evaluation of FLEXFUZZ by addressing the following research questions:

RQ1 How does FLEXFUZZ perform compared to state-of-the-art parallel fuzzers?

RQ2 What are the contributions of the main components of FLEXFUZZ?

A. Experimental Setup

1) Benchmark: Traditional benchmarks often target simple programs, with over half containing under 10,000 basic blocks and 20,000 branches, and are less suitable for parallel fuzzing evaluation. Therefore, we select the most complex programs from FuzzBench [33], OSS-Fuzz [5], and UNIFUZZ [34], complemented by widely used large-scale software like Vim [35] and ImageMagick [36]. Table III provides details.

2) Environment: Experiments are conducted on five machines, each with two Intel Xeon Platinum 8380 processors (160 virtual cores total) using Docker on Ubuntu 20.04. For each test, a parallel fuzzing system is allocated 10 cores (workers) and runs for 24 hours. Each experiment is repeated 10 times to ensure reliability.

3) Baseline: We evaluate FLEXFUZZ against six parallel fuzzers: AFL++ and K-Scheduler in parallel mode (abbreviated as AFL++ and K-Scheduler), PAFL, AFL-EDGE, AFLTeam, and autofz. K-Scheduler is included for its graph-based scheduling, which shares similarities with FLEXFUZZ. Due to their task allocation mechanisms, PAFL, AFL-EDGE, and AFLTeam are selected as primary baselines. Since PAFL is not open source, we have implemented the PAFL approach in AFL++ from the published description. Similarly, K-Scheduler, AFLTeam, and AFL-EDGE were adapted to AFL++ for fair evaluation. Although autofz does not explicitly target task duplication, it is included for dynamic fuzzer composition based on performance. In contrast, μFUZZ is excluded due to persistent memory errors and failure to record meaningful results like crashes.

B. Effectiveness of FLEXFUZZ

We assess effectiveness based on two metrics: code coverage and detected vulnerabilities.

1) Branch Coverage: As presented in Table IV, FLEXFUZZ achieves the highest average coverage across most programs, outperforming AFL++ by 21.04%. Other parallel fuzzers show less than a 5% improvement or even underperform compared to AFL++. On complex programs like vim, magick, and ffmpeg, FLEXFUZZ exceeds AFL++ by over 35%. While AFL-EDGE, AFLTeam, and PAFL achieve competitive results on specific large-scale benchmarks, FLEXFUZZ consistently excels. Additionally, introducing high computational overhead significantly reduces performance, as observed with K-Scheduler and autofz. They often struggle with larger ones, occasionally failing to execute correctly. On pdftotext, FLEXFUZZ occasionally achieves exceptionally high coverage (50,000+ and 26,000+), though with a low probability. For fairness, these outliers are excluded from Table IV. Further analysis reveals that this spike stems from specific seeds resolving critical branch conditions, which FLEXFUZZ generates by targeting boundary areas. To see how these results develop over time, Figure 6 traces coverage across 24 hours: FLEXFUZZ establishes an early lead on essentially every target, and the margin typically widens on larger programs—for example, on *vim* it maintains that lead throughout and finishes 57.04% above AFL++ (Table IV).

2) Unique Vulnerability: We recompiled programs with AddressSanitizer (ASan) [37] and identified a crash as a unique vulnerability if the top three code locations in the ASan stack trace were distinct. This stack hashing method, proposed by Klees et al. [38], is widely used in practice but may overcount vulnerabilities. As shown in Table V, the traditional vulnerability filtering method exhibits significant inaccuracies, rendering it unsuitable for evaluating parallel fuzzing systems. For instance, pdftotext and sqlite3 reported over 100 vulnerabilities, an excessive and unreliable result. Even though counts seemed more reasonable in nm-new and ffmpeg, our verification revealed that they were overcounted. To refine the results, we applied additional filtering, leveraging ChatGPT 3.5 to analyze crash reports and performing manual verification to remove duplicates with identical error types or closely related locations. Table V presents the number of unique vulnerabilities after this deduplication. No crashes were found in tcpdump, vim, or magick, so they were excluded. Overall, FLEXFUZZ identified 107 unique bugs, 27 more than AFL++, while others performed similarly to AFL++. Figure 7 reports bugs that are absent from public CVE records, with the matching procedure described online [28]. In the heat map, FLEXFUZZ shows broad strength across major classes, including 40 segmentation faults, 34 heap buffer overflows, and 12 stack overflows. In the intersection view, FLEXFUZZ contributes 17 exclusive findings and also appears in many shared intersections, which shows that it both uncovers defects missed by peers and covers those that peers detect. Details on whether these bugs were previously known are provided on the project site [28].

C. Ablation Study

We evaluated each component through targeted experiments and relevant metrics. FLEXFUZZ$_T$ applies task assignment with fixed scheduling, while FLEXFUZZ$_{TD}$ uses task allocation with adaptive scheduling but without directed exploration.

1) Basic Metric: Table VI illustrates the improvements achieved by the main components of FLEXFUZZ. With a boundary-sensitive task assignment strategy and hourly task updates, FLEXFUZZ$_T$ outperforms AFL++ in coverage and bug discovery, even equipped with fixed interval scheduling.

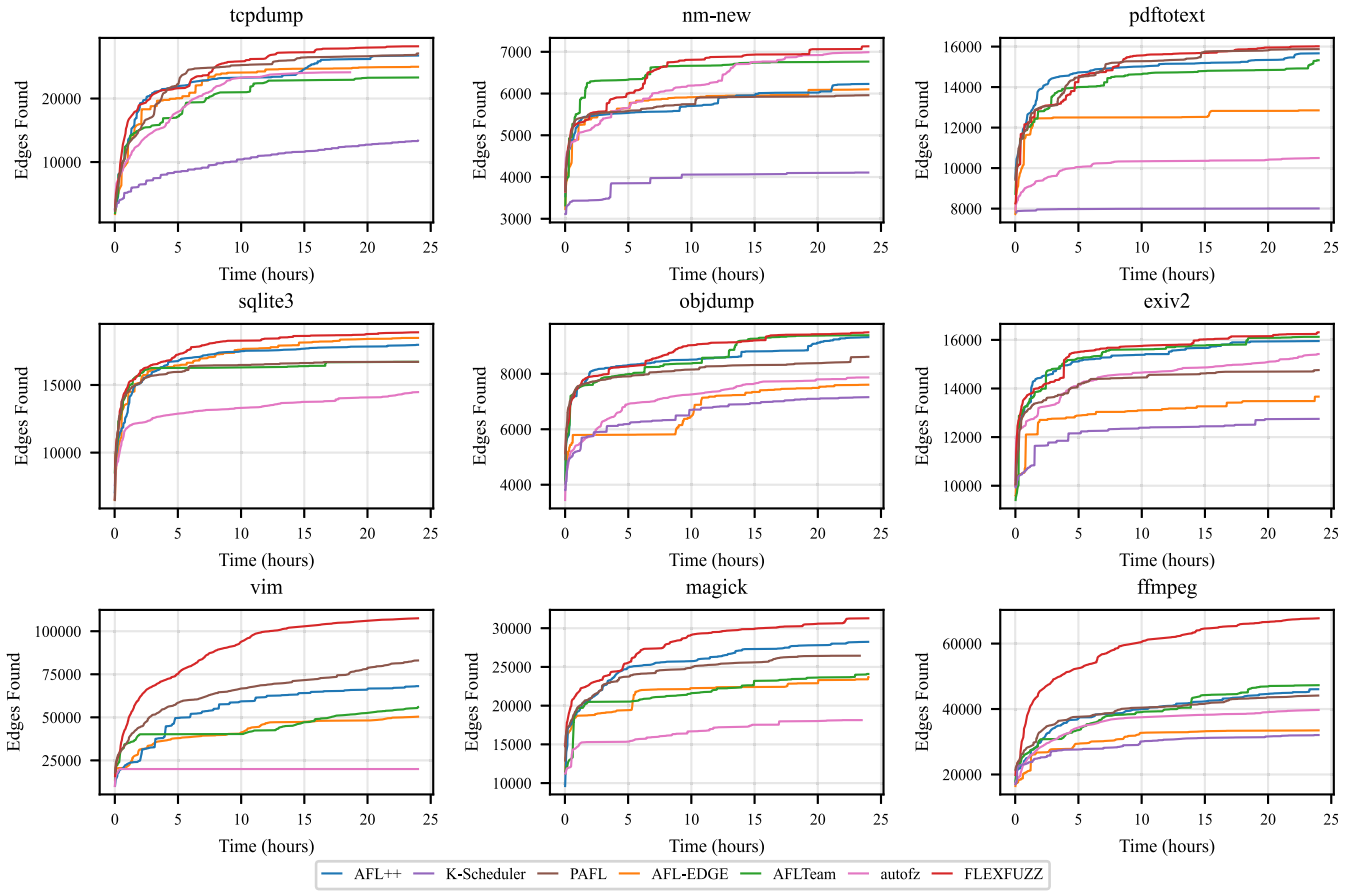


Fig. 6. Growth of edge coverage over 24 hours on our real-world benchmarks.

TABLE IV
THE AVERAGE COVERED BRANCHES AND PERCENTAGE INCREMENTS OVER AFL++ ARE REPORTED ACROSS TEN TRIALS

Program	AFL++	K-Scheduler		PAFL		AFL-EDGE		AFLTeam		autofz		FLEXFUZZ	
	Number	Number	Increase	Number	Increase	Number	Increase	Number	Increase	Number	Increase	Number	Increase
tcpdump	25,775.90	13,909.69	-46.04%	25,651.00	-0.48%	26,200.10	1.65%	23,453.70	-9.01%	24,106.67	-6.48%	**26,811.10**	**4.02%**
nm-new	6,169.20	4,107.10	-33.43%	6,169.20	0.00%	6,100.00	-1.12%	6,574.00	6.56%	7,029.40	13.94%	**7,059.40**	**14.43%**
pdftotext	15,424.20	7,980.94	-48.26%	15,457.40	0.22%	14,934.50	-3.17%	15,336.40	-0.57%	10,676.30	-30.78%	**15,787.40**	**2.35%**
sqlite3	17,651.00	-	-	17,836.80	1.05%	17,543.50	-0.61%	16,992.10	-3.73%	14,906.22	-15.55%	**18,590.80**	**5.32%**
objdump	9,072.60	7,158.43	-21.10%	8,589.10	-5.33%	8,711.40	-3.98%	9,218.90	1.61%	7,888.60	-13.05%	**9,584.80**	**5.65%**
exiv2	16,099.40	12,743.90	-20.84%	16,055.60	-0.27%	16,022.90	-0.48%	15,252.40	-5.26%	15,426.90	-4.18%	**16,432.70**	**2.07%**
vim	63,356.50	-	-	78,884.00	24.51%	56,509.50	-10.81%	60,233.60	-4.93%	19,919.70	-68.56%	**99,497.10**	**57.04%**
magick	23,484.10	-	-	22,560.00	-3.94%	25,165.00	7.16%	25,989.80	10.67%	18,031.11	-23.22%	**32,505.30**	**38.41%**
ffmpeg	43,310.50	32,053.00	-25.99%	44,979.50	3.85%	49,861.20	15.12%	46,092.20	6.42%	39,691.30	-8.36%	**69,309.40**	**60.03%**
AVG	24,482.60	12,992.18	-32.61%	26,242.51	2.18%	24,560.90	0.42%	24,349.23	0.20%	17,519.58	-17.36%	**32,842.00**	**21.04%**

"-" denotes compile or execution errors preventing K-Scheduler support.

Further improvements arising from dynamic task scheduling and focused exploration demonstrate the effectiveness of each component in optimizing parallel fuzzing. Although the degradation relative to the full FLEXFUZZ remains small (around 2–3%), this indicates that both components make tangible contributions. We provide a more detailed breakdown and discussion of these effects in the following analysis section.

2) Task Assignment: We introduce the Average Jaccard Index to assess the impact of our task assignment. This metric measures the similarity between two sets. For two sets A and B, the Jaccard Index $J(A, B)$ is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{7}$$

We calculate the Jaccard Index for execution branch sets across newly generated seeds between two engines to quantify exploration redundancy. For multiple engines, we compute the Jaccard Index for each pair and take the average. For n engines,

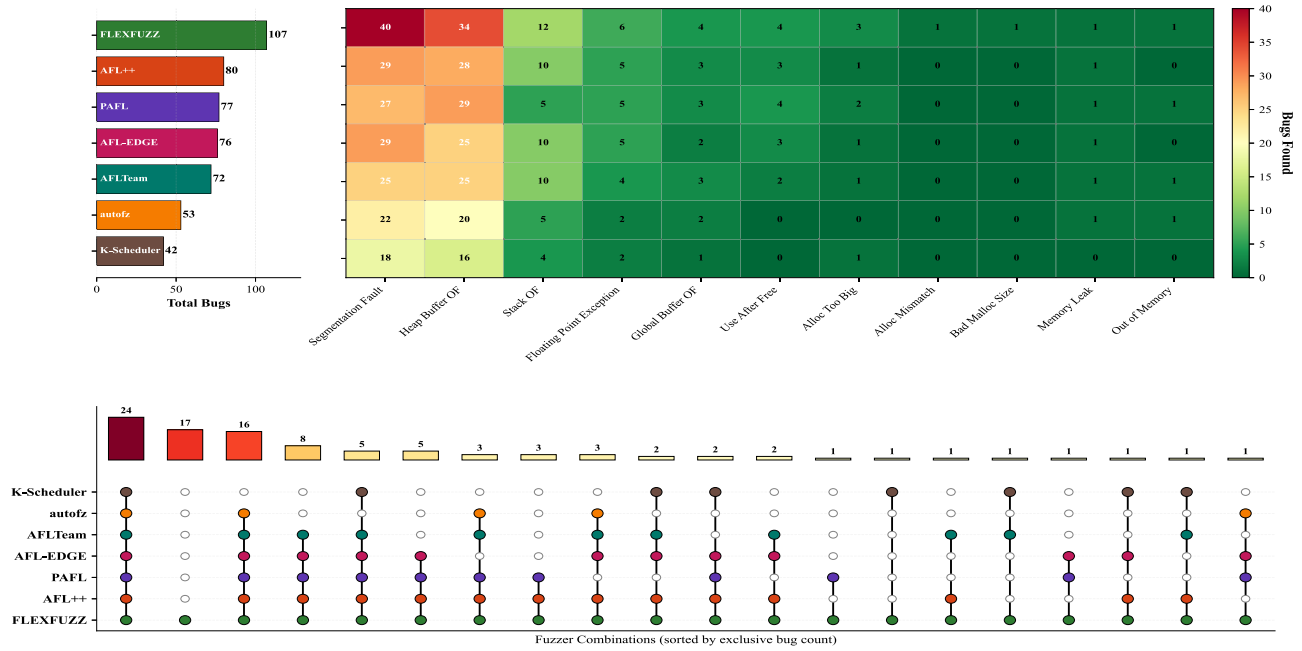


Fig. 7. Top: heat map with fuzzers as rows and bug classes as columns; darker cells indicate more bugs for that fuzzer in that class, and the total at the end of each row summarizes that fuzzer's overall yield. Bottom: intersection (UpSet) plot; each bar counts bugs found only by the exact combination of fuzzers shown by the filled dots beneath it. A bar with a single filled dot indicates findings unique to that fuzzer, while a bar with several filled dots indicates findings shared by those fuzzers and by no others. Together, the panels show the distribution by class and the concrete results between unique and shared findings.

TABLE V

THE NUMBER OF UNIQUE VULNERABILITIES AFTER DEDUPLICATION ACROSS TEN TRIALS

Program	AFL++		K-Scheduler		PAFL		AFL-EDGE		AFLTeam		autofz		FLEXFUZZ	
	Top3	Filter	Top3	Filter	Top3	Filter	Top3	Filter	Top3	Filter	Top3	Filter	Top3	Filter
nm-new	0	0	0	0	1	1	1	1	1	1	0	0	**3**	**2**
pdftotext	83	28	50	15	85	25	82	26	93	26	60	20	**98**	**34**
sqlite3	86	15	-	-	141	14	127	14	**387**	12	80	7	201	**17**
objdump	19	7	10	4	**25**	9	16	6	14	6	12	5	20	**14**
exiv2	**92**	27	62	22	91	26	90	25	66	25	60	20	82	**33**
ffmpeg	5	3	3	1	7	2	8	4	3	2	3	1	**10**	**7**
SUM	285	80	125	42	350	77	324	76	**564**	72	215	53	414	**107**

"Top3" refers to filtering crashes by the top three stack frames.
"Filter" refers to further filtering bugs by LLM and manual verification.

TABLE VI

COVERAGE AND VULNERABILITIES ACROSS TEN TRIALS FOR ABLATION STUDY. "INCREASE" INDICATES THE IMPROVEMENT OVER AFL++, WHILE "DEGRADE" SHOWS THE PERFORMANCE DROP RELATIVE TO FLEXFUZZ

Program	FLEXFUZZ$_T$					FLEXFUZZ$_{TD}$				
	Number	Increase	Degrade	Top3	Filter	Number	Increase	Degrade	Top3	Filter
tcpdump	26,454.00	2.63%	-1.33%	0	0	26,689.60	3.54%	-0.45%	0	0
nm-new	6,940.60	12.50%	-1.68%	1	1	7,000.50	13.48%	-0.83%	1	1
pdftotext	15,505.10	0.52%	-1.79%	83	30	15,739.20	2.04%	-0.31%	80	31
sqlite3	18,487.40	4.74%	-0.56%	222	16	18,567.90	5.19%	-0.12%	214	17
objdump	9,089.70	0.19%	-5.17%	14	10	9,424.70	3.88%	-1.67%	18	10
exiv2	16,256.50	0.98%	-1.07%	84	28	16,387.60	1.79%	-0.27%	83	30
vim	95,604.10	50.90%	-3.91%	0	0	97,720.50	54.24%	-1.79%	0	0
magick	31,083.70	32.36%	-4.37%	0	0	31,377.30	33.61%	-3.47%	0	0
ffmpeg	64,475.60	48.87%	-6.97%	7	6	64,575.70	49.10%	-6.83%	6	6
AVG/SUM	31544.08	17.08%	-2.98%	411	91	31942.56	18.54%	-1.75%	402	95

the Average Jaccard Index is defined as:

$$\text{Average Jaccard Index} = \frac{1}{\binom{n}{2}} \sum_{1 \le i < j \le n} J(E_i, E_j) \qquad (8)$$

The sets E_i and E_j represent execution branches covered by new seeds from engines i and j, excluding initial and synchronized seeds. The binomial coefficient $\binom{n}{2}$ denotes the number of unique engine pairs. Since our implementation is based on AFL++ and other tools show only minor improvements, we introduce this metric to evaluate task assignment effectiveness. A lower Average Jaccard Index means less overlap among workers and thus greater exploration diversity. Our aim here is to examine whether FLEXFUZZ$_T$ consistently maintains lower redundancy than AFL++, showing that workers complement rather than duplicate each other.

As FLEXFUZZ$_T$ initiates a new exploration round every hour, we compute the Average Jaccard Index at rounds 2, 4, 8, and 16. For consistency, we run AFL++ with initial seeds from the corresponding rounds, repeat each round ten times, and calculate the Average Jaccard Index after one hour of parallel fuzzing. Currently, no standard metrics exist for measuring

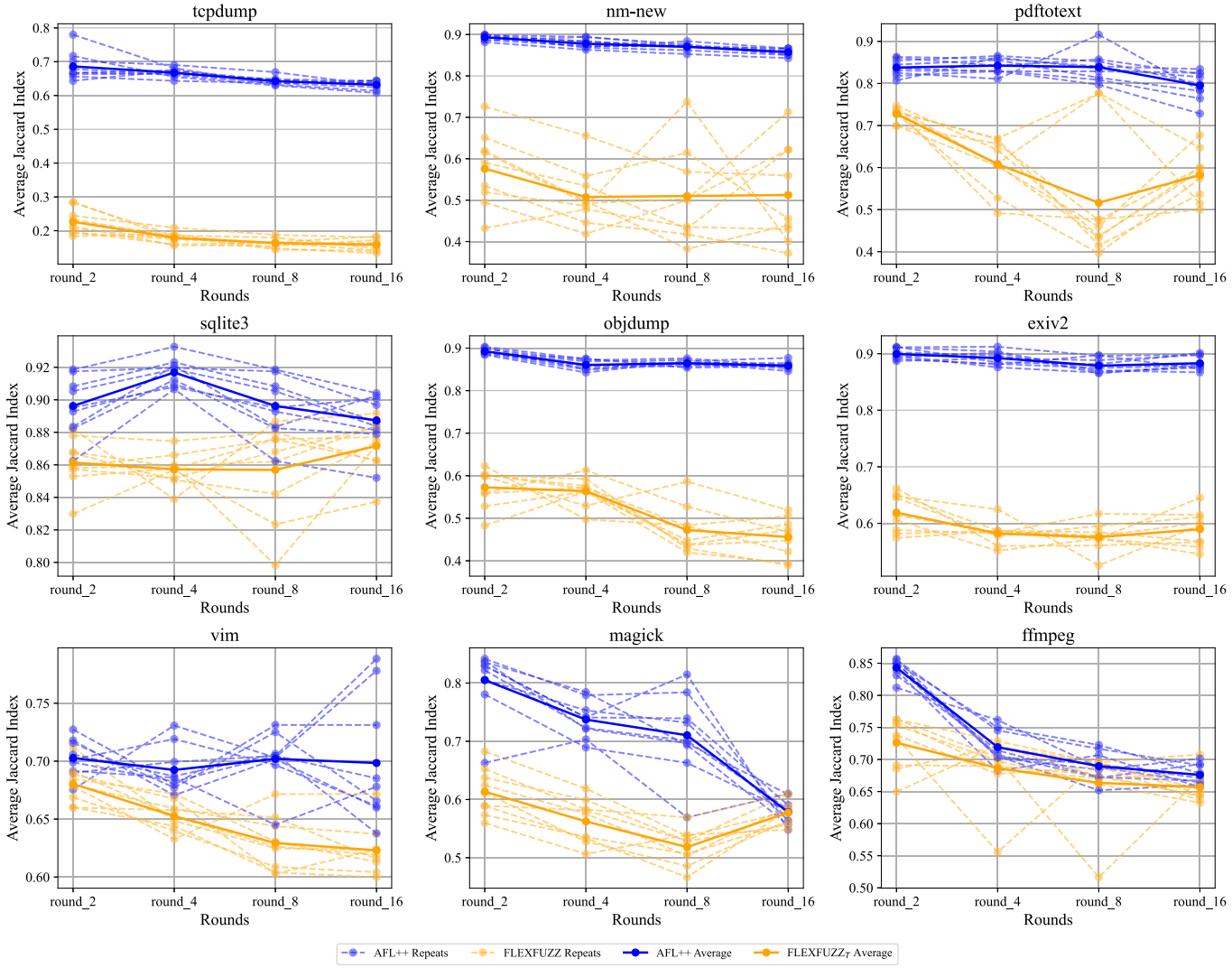


Fig. 8. The line chart depicts the Average Jaccard Index for the covered edges of new seeds across workers.

task exploration redundancy; a detailed discussion of the task conflicts metric is provided online [28].

Figure 8 presents the statistical results. Among the nine targets, FLEXFUZZ$_T$ consistently achieves a lower Average Jaccard Index than AFL++, emphasizing the effectiveness of our task assignment strategy in reducing exploration redundancy. Consequently, FLEXFUZZ$_T$ explores more execution paths and more efficiently. Fixed-interval task updates maintain a low Average Jaccard Index over time. However, these updates do not align with coverage growth rates. In programs with extensive branching, a one-hour update interval leads to higher duplication. In contrast, in programs with fewer branches, such as pdftotext and objdump, redundancy remains low but minimal overall coverage improvement (see Table VI). These findings highlight the need for adaptive task scheduling.

3) Task Scheduling: We analyzed coverage changes and task update intervals with FLEXFUZZ$_{TD}$ to illustrate its dynamic scheduling mechanism. Figure 9 shows that exploration efficiency varies across trials for the same program due to fuzzing randomness. To reduce clutter, we selected three representative sets based on the highest, moderate, and lowest update counts. FLEXFUZZ$_{TD}$ adjusts task updates according to coverage growth trends. The trends differ notably across programs: vim and ffmpeg exhibit steady growth over 24 hours, prompting frequent updates, while tcpdump, nm-new, and pdftotext plateau after 40,000 seconds, leading to fewer updates and extended exploration in challenging areas.

These within-run traces explain how the policy reacts; to evaluate its intended effect across trials, we next quantify *stability*. As the objective of *Adaptive Scheduling Based on Feedback* is to improve stability, we assess this property with the Coefficient of Variation (CV). For each target, let μ denote the mean outcome across independent trials and σ the corresponding standard deviation; we report:

$$\text{CV} = 100\% \times \frac{\sigma}{\mu} \tag{9}$$

CV is dimensionless and summarizes relative dispersion; smaller values indicate more repeatable outcomes at a fixed time budget. In Figure 10, across eight benchmarks, FLEXFUZZ$_{TD}$ exhibits lower CV than FLEXFUZZ$_T$ on

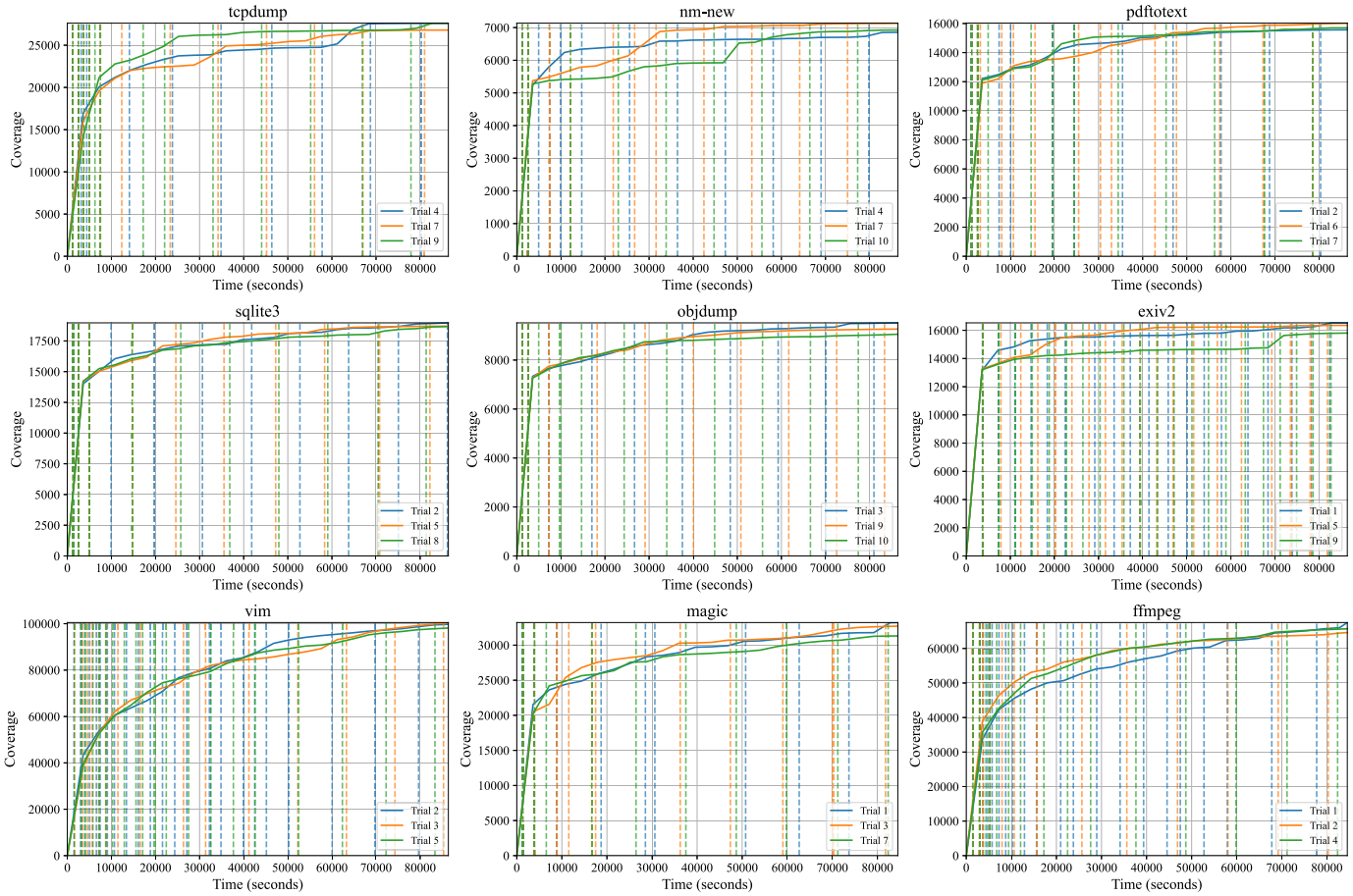


Fig. 9. Coverage changes over time are marked with task update points for each trial. The trial ID represents the repeat ID from the ten evaluations of FLEXFUZZ$_{TD}$. Vertical dotted lines indicate individual task updates.

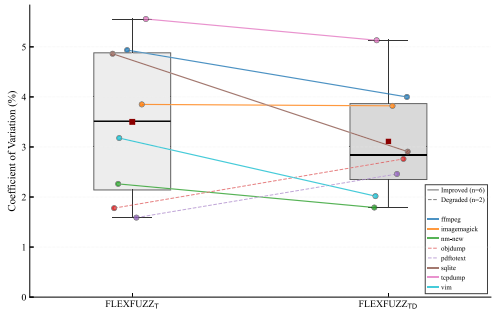


Fig. 10. Dumbbell plot comparing coefficient of variation (CV) between FLEXFUZZ$_T$ and FLEXFUZZ$_{TD}$. Each dot shows the CV (%) for one target program. Gray boxes indicate the interquartile range (IQR) of CV values across all targets for each method. Red squares mark the median CV. Solid connectors indicate FLEXFUZZ$_{TD}$ has lower CV than FLEXFUZZ$_T$ for that target; dotted connectors indicate equal or higher CV. Lower CV values represent better stability.

most targets, with only minor differences on the remainder. The observed reduction in CV indicates that feedback-driven task scheduling reduces between-trial variance by reallocating effort away from saturated regions and alleviating stagnation earlier, yielding more stable results.

4) Task Exploration: We evaluate multi-target distance-guided exploration by analyzing task exploration in a representative middle round. Table VII summarizes the boundary and target basic blocks assigned to each engine, with task sizes ranging from hundreds to thousands of boundary basic blocks. SO_{TBD} and SO_{TTG} represent original seeds triggering BB_{TBD} and BB_{TTG}, while SF_{TBD} and SF_{TTG} denote new seeds triggering these blocks after fuzzing. R_{orig} and R_{fuzz} indicate the ratios of seeds triggering BB_{TTG} to seeds triggering BB_{TBD} before and after fuzzing. The results show a substantial increase in the proportion of new seeds that trigger target basic blocks, rising from 14.41% to 45.51% on average.

TABLE VII

SEED COUNTS AND PROPORTIONS TRIGGERED TASK BOUNDARY AND TARGET BASIC BLOCKS BEFORE AND AFTER FUZZING, ALONG WITH RELATED BASIC BLOCK COUNTS ACROSS TEN TRIALS

Program	BB_{TBD}	BB_{TTG}	SO_{TBD}	SO_{TTG}	R_{orig}	SF_{TBD}	SF_{TTG}	R_{fuzz}
tcpdump	339.71	80.50	3,213.33	77.70	2.43%	313.21	73.21	23.19%
nm-new	215.86	67.61	770.59	78.01	10.22%	74.86	40.01	53.22%
pdftotext	311.07	92.78	1,595.35	392.64	24.66%	254.92	129.22	50.93%
sqlite3	652.51	209.34	1,120.98	107.48	9.62%	231.88	114.76	49.69%
objdump	241.11	76.41	962.36	131.98	13.79%	132.81	64.26	48.32%
exiv2	971.18	311.78	1,085.64	311.00	28.56%	157.88	106.79	67.69%
vim	3,357.32	1,120.46	9,700.08	247.31	2.62%	1,314.23	278.26	22.10%
magick	2,398.92	798.22	2,704.37	633.09	23.38%	625.16	295.88	47.04%
ffmpeg	1,702.19	533.86	4,993.96	717.56	14.41%	546.11	254.22	47.44%
AVG	1,132.21	365.66	2,905.18	299.64	14.41%	405.67	150.73	45.51%

The increase demonstrates that FLEXFUZZ effectively directs exploration toward challenging target nodes.

VII. LIMITATION AND DISCUSSION

A. Task Allocation

Although boundary-sensitive task allocation identifies critical regions as coverage grows, it does not directly prioritize vulnerability triggering. Previous research has utilized fine-grained features, such as memory operations [39], bug-detection potential [31], and various program properties [40], [41], [42], to enhance vulnerability discovery. Integrating these insights could refine program state analysis, critical code area assessments, and state-sensitive task allocation for better vulnerability mining.

B. Task Exploration

FLEXFUZZ employs a targeted task exploration mechanism through seed selection and energy adjustment. However, existing studies [3], [42], [43], [44], [45] indicate that mutation operator distribution and types significantly affect fuzzing performance. To enhance FLEXFUZZ, we propose incorporating advanced mutation strategies while maintaining compatibility with task exploration.

C. Scalability in Distributed Scenarios

FLEXFUZZ is implemented as a multiprocess program for single-machine operation. FLEXFUZZ could be extended to run separately on multiple machines in a distributed scenario, synchronizing information via a remote procedure call mechanism. Each machine could utilize distinct databases to store information. However, slow network I/O presents a significant obstacle requiring further research.

VIII. RELATED WORK

A. Directed Fuzzing

Although Parmesan [32] and SAVIOR [31] handle seed distances for multiple targets, they inherit AFLGo's limitation: encoding basic-block–to-target distance as a single scalar, thereby losing precision. FLEXFUZZ improves on this by calculating distances using a real-time control flow graph and selecting the minimum distance. While Titan [25] distinguishes correlations between targets, and FISHFUZZ [26] proposes a function-level multi-target distance metric with target-independent precision, both struggle with dynamic target determination and real-time exploration changes due to high overhead. Nevertheless, their target filtering and multi-target mutation strategies inspire further improvements to FLEXFUZZ.

B. Parallel Fuzzing

Single-machine parallel fuzzing leverages multi-core systems to run multiple instances concurrently. AFL and AFL++ provide foundational parallel modes, with improvements such as reducing synchronization costs [46] and introducing microservice architectures [14] to enhance performance. Building on these, methods like static task division [13] and dynamic task allocation strategies [15], [16], [17] optimize resource use by alleviating task conflicts. FLEXFUZZ further refines task allocation by targeting promising areas for incremental and focused exploration, addressing limitations in prior approaches. Ensemble parallel fuzzing optimizes resources by integrating various fuzzers. EnFuzz relies on expert input for diverse prototype selection, while Cupid automates compatibility prediction through an offline complementarity indicator. Moreover, autofz enhances them by collecting trends to adjust fuzzer composition. Although FLEXFUZZ currently operates in a single-prototype framework, its performance could benefit from incorporating autofz. Distributed fuzzing addresses the scalability limitations of single-machine setups but introduces challenges in task coordination. The lightweight framework of Roving [47] and the client-server model of P-Fuzz improve distributed setups but face issues like frequent file transfers and task conflicts. Advanced solutions like CollabFuzz [48] enhance coordination through efficient scheduling and test case distribution. While FLEXFUZZ's containerized design is compatible with CollabFuzz, further refinements are necessary to overcome network overhead and synchronization challenges. Parallel fuzzing also applies to protocol-centric domains. SPFuzz [49] constructs protocol state and data models to enhance task allocation for automotive vehicle protocols, whereas MPFuzz [50] targets IoT messaging protocols via collaborative packet generation. This protocol-aware parallelism represents a promising direction for future research and practice.

IX. CONCLUSION

To mitigate shortcomings of task allocation and exploration within existing parallel fuzzing systems, we introduce a boundary-sensitive task allocation scheme and a multi-target distance-guided exploration method, implementing them in our proposed parallel fuzzing system, FLEXFUZZ. Experiments show significant improvements. This study highlights the need for efficient task exploration in parallel fuzzing and paves the way for future advancements and distributed applications.

REFERENCES

[1] Y. Liu and W. Meng, "DSFuzz: Detecting deep state bugs with dependent state exploration," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2023, pp. 1242–1256, doi: 10.1145/3576915.3616594.

[2] X. Zhu, S. Zhang, C. Li, S. Wen, and Y. Xiang, "Fuzzing Android native system libraries via dynamic data dependency graph," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 3733–3744, 2024, doi: 10.1109/TIFS.2024.3369479.

[3] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "DARWIN: Survival of the fittest fuzzing mutators," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–17.

[4] J. Li, S. Li, G. Sun, T. Chen, and H. Yu, "SNPSFuzzer: A fast greybox fuzzer for stateful network protocols using snapshots," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 2673–2687, 2022, doi: 10.1109/TIFS.2022.3192991.

[5] Googlea. (2016). *Oss-Fuzz*. Accessed: Oct. 14, 2025. [Online]. Available: https://github.com/google/oss-fuzz

[6] Google. (2019). *Clusterfuzz*. Accessed: Oct. 14, 2025. [Online]. Available: https://github.com/google/clusterfuzz

[7] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. 14th USENIX Workshop Offensive Technol. (WOOT)*, Aug. 2020, pp. 1–11. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[8] Y. Chen et al., "EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur. 19)*, Aug. 2018, pp. 1967–1983. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

[9] E. Güler et al., "Cupid: Automatic fuzzer selection for collaborative fuzzing," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2020, pp. 360–372, doi: 10.1145/3427228.3427266.

[10] Y.-F. Fu, J. Lee, and T. Kim, "Autofz: Automated fuzzer composition at runtime," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur. 23)*, Aug. 2023, pp. 1901–1918. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/fu-yu-fu

[11] C. Song, X. Zhou, Q. Yin, X. He, H. Zhang, and K. Lu, "P-fuzz: A parallel grey-box fuzzing framework," *Appl. Sci.*, vol. 9, no. 23, p. 5100, Nov. 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/23/5100

[12] X. Zhou et al., "UltraFuzz: Towards resource-saving in distributed fuzzing," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2394–2412, Apr. 2023, doi: 10.1109/TSE.2022.3219520.

[13] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "PAFL: Extend fuzzing optimizations of single mode to industrial parallel mode," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Oct. 2018, pp. 809–814, doi: 10.1145/3236024.3275525.

[14] Y. Chen, R. Zhong, Y. Yang, H. Hu, D. Wu, and W. Lee, "FUZZ: Redesign of parallel fuzzing using microservice architecture," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur. 23)*, Aug. 2023, pp. 1325–1342. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/chen-yongheng

[15] Y. Wang, Y. Zhang, C. Pang, P. Li, N. Triandopoulos, and J. Xu, "Facilitating parallel fuzzing with mutually-exclusive task distribution," in *Proc. Secur. Privacy Commun. Netw.*, 2021, pp. 185–206.

[16] V.-T. Pham, M.-D. Nguyen, Q.-T. Ta, T. Murray, and B. I. P. Rubinstein, "Towards systematic and dynamic task allocation for collaborative parallel fuzzing," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 1337–1341.

[17] T. Gu et al., "Group-based corpus scheduling for parallel fuzzing," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2022, pp. 1521–1532, doi: 10.1145/3540250.3560885.

[18] D. She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 2194–2211.

[19] H. Rong, W. You, X. Wang, and T. Mao, "Toward unbiased multiple-target fuzzing with path diversity," in *Proc. 33rd USENIX Secur. Symp. (USENIX Secur. 24)*, Aug. 2023, pp. 2475–2492. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/rong

[20] D. She, A. Storek, Y. Xie, S. Kweon, P. Srivastava, and S. Jana, "FOX: Coverage-guided fuzzing as online stochastic control," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Dec. 2024, pp. 765–779.

[21] M. Zalewski. (2025). *AFL—American Fuzzy Lop*. Google. Accessed: Aug. 28, 2025. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[22] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2329–2344, doi: 10.1145/3133956.3134020.

[23] H. Chen et al., "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 2095–2108.

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

[25] H. Huang, P. Yao, H.-C. Chiu, Y. Guo, and C. Zhang, "Titan: Efficient multi-target directed greybox fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 1849–1864.

[26] H. Zheng et al., "FISHFUZZ: Catch deeper bugs by throwing larger nets," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur. 23)*, 2023, pp. 1343–1360.

[27] H. Liang, X. Yu, X. Cheng, J. Liu, and J. Li, "Multiple targets directed greybox fuzzing," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 1, pp. 325–339, Jan. 2024.

[28] F. Team. (2025). *FLEXFUZZ*. Accessed: Oct. 14, 2025. [Online]. Available: https://diewufeihong.github.io/FLEXFUZZ/

[29] Wikipedia Contributors. (2024). *TCP Congestion Control—Wikipedia, The Free Encyclopedia*. Accessed: Aug. 9, 2024. [Online]. Available: https://en.wikipedia.org/wiki/TCP_congestion_control

[30] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 2169–2182, doi: 10.1145/3460120.3484596.

[31] Y. Chen et al., "SAVIOR: Towards bug-driven hybrid testing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1580–1596.

[32] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided greybox fuzzing," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur. 20)*, 2020, pp. 2289–2306.

[33] G. Inc. (2025). *Fuzzbench: A Free Fuzzer Benchmarking Service*. Accessed: Oct. 14, 2025. [Online]. Available: https://github.com/google/fuzzbench

[34] Y. Li et al., "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *Proc. USENIX Secur. Symp.*, 2021, pp. 2777–2794. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[35] Bram Moolenaar and the Vim Contributors. (2024). *Vim*. Accessed: Oct. 14, 2025. [Online]. Available: https://www.vim.org/

[36] ImageMagick Studio LLC. (2024). *Imagemagick*. Accessed: Oct. 14, 2025. [Online]. Available: https://imagemagick.org/

[37] Google. (Jan. 2024). *AddressSanitizer*. Accessed: Oct. 14, 2025. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[38] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Oct. 2018, pp. 2123–2138, doi: 10.1145/3243734.3243804.

[39] Y. Wang et al., "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17. [Online]. Available: https://api.semanticscholar.org/CorpusID:211268394

[40] R. Liang et al., "Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing," *IEEE Trans. Inf. Forensics Security*, vol. 20, pp. 2157–2170, 2025, doi: 10.1109/TIFS.2025.3537827.

[41] C. Lyu et al., "SLIME: Program-sensitive energy allocation for fuzzing," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2022, pp. 365–377, doi: 10.1145/3533767.3534385.

[42] S. Gan et al., "GREYONE: Data flow sensitive fuzzing," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur. 20)*, Aug. 2020, pp. 2577–2594. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[43] C. Lyu et al., "EMS: History-driven mutation for coverage-based fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2022, pp. 24–28.

[44] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[45] C. Lyu et al., "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. USENIX Secur. Symp.*, 2019, pp. 1949–1966. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[46] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2313–2328, doi: 10.1145/3133956.3134046.

[47] R. Healey. (2022). *Roving: A Traveling Salesman Problem Solver With a Focus on Heuristics*. [Online]. Available: https://github.com/richo/roving

[48] S. Österlund et al., "CollabFuzz: A framework for collaborative fuzzing," in *Proc. 14th Eur. Workshop Syst. Secur.*, Apr. 2021, pp. 1–7, doi: 10.1145/3447852.3458720.

[49] J. Yu, Z. Luo, F. Xia, Y. Zhao, H. Shi, and Y. Jiang, "SPFuzz: Stateful path based parallel fuzzing for protocols in autonomous vehicles," in *Proc. 61st ACM/IEEE Design Autom. Conf.*, Jun. 2024, pp. 1–6.

[50] Z. Luo et al., "Parallel fuzzing of IoT messaging protocols through collaborative packet generation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 43, no. 11, pp. 3431–3442, Nov. 2024.

Hong Liang received the bachelor's degree from the Huazhong University of Science and Technology. She is currently pursuing the Ph.D. degree with Zhejiang University. Her research interests include software and system security.

Yijia Guo received the bachelor's degree from Zhejiang Normal University, where he is currently pursuing the Ph.D. degree. His research interests include software and system security.

Haotian Wu received the bachelor's degree in software engineering from Dalian University of Technology. He is currently pursuing the master's degree with Zhejiang University. His research interests include software and system security.

Yifan Xia received the bachelor's degree from the School of Cyber Science and Engineering, Huazhong University of Science and Technology, in 2022. He is currently pursuing the Ph.D. degree with the Department of Control Science and Engineering, Zhejiang University. His research interests include software security and program analysis. More information about him can be found at: https://evanxiaa.github.io/

Yi Xiang received the bachelor's degree from Wuhan University. She is currently pursuing the Ph.D. degree with Zhejiang University. Her research interests include software and system security.

Xiantao Jin received the bachelor's degree from Wuhan University of Technology in 2008. He was a Senior Engineer at the Fifth Research Institute of Electronics, Ministry of Industry and Information Technology (MIIT). He is currently the Deputy Group Leader of Preliminary Screening for the Open-Source Vulnerability Database at the OpenAtom Foundation. His research interests include binary fuzz testing, binary program static analysis, and firmware emulation. As a key contributor, he has participated in two National Key Research and Development Programs of China.

Hao Peng received the Ph.D. degree in computer science and technology from Shanghai Jiao Tong University in 2012. He is currently a Full Professor at the School of Computer Science and Technology, Zhejiang Normal University. He has published more than 100 papers in prestigious journals and conferences. His research interests include AI security, the IoT security, software and system security, data-driven security, big data mining, and analysis. He served as a program committee member for more than ten international conferences.

Xuhong Zhang received the Ph.D. degree in computer engineering from the University of Central Florida in 2017. He is currently a ZJU 100-Young Professor with the School of Software Technology, Zhejiang University. He has authored over 20 publications in premier journals and conferences, such as IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE S&P, USENIX Security, ACM CCS, NDSS, and VLDB. His research interests include distributed big data and AI systems, big data mining and analysis, data-driven security, and AI and security.

Shouling Ji (Member, IEEE) received the B.S. (Hons.) and M.S. degrees in computer science from Heilongjiang University, the Ph.D. degree in electrical and computer engineering from Georgia Institute of Technology, and the Ph.D. degree in computer science from Georgia State University. He is currently a Qiushi Distinguished Professor at the College of Computer Science and Technology, Zhejiang University, and an Adjunct Research Faculty at the School of Electrical and Computer Engineering, Georgia Institute of Technology. His current research interests include data-driven security and privacy, AI security and software, and system security. He is a member of ACM and a Senior Member of CCF, and was the Membership Chair of the IEEE Student Branch at Georgia State University (2012–2013). He was a Research Intern at the IBM T. J. Watson Research Center. He was a recipient of the 2012 Chinese Government Award for Outstanding Self-Financed Students Abroad and ten best/outstanding paper awards, including ACM CCS 2021.