Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.065162

ARTICLE

RBZZER: A Directed Fuzzing Technique for Efficient Detection of Memory Leaks via Risk Area Analysis

Xi Peng, Peng Jia*, Ximing Fan and Jiayong Liu*

School of Cyber Science and Engineering, Sichuan University, Chengdu, 610065, China

*Corresponding Authors: Peng Jia. Email: pengjia@scu.edu.cn; Jiayong Liu. Email: ljy@scu.edu.cn

Received: 05 March 2025; Accepted: 27 May 2025; Published: 30 July 2025

ABSTRACT: Memory leak is a common software vulnerability that can decrease the reliability of an application and, in severe cases, even cause program crashes. If there are intentionally triggerable memory leak vulnerabilities in a program, attackers can exploit these bugs to launch denial-of-service attacks or induce the program to exhibit unexpected behaviors due to low memory conditions. Existing fuzzing techniques primarily focus on improving code coverage, and specialized fuzzing techniques for individual memory-related defects like uncontrolled memory allocation do not address memory leak vulnerabilities. MemLock is the first fuzzing technique to address memory consumption vulnerabilities including memory leakage. However, the coverage-centric guidance mechanism of MemLock introduces a degree of aimlessness in the testing process, that results in low seed quality and slow bug exposure speed. To address this issue, we propose a risk areas guidance-based fuzzing technique called RBZZER. First, RBZZER retains MemLock's memory consumption-guided mechanism and introduces a novel distance-guided approach to expedite the arrival of fuzzing at the potential memory areas. Second, we introduce a new seed scheduling strategy called risk areas-based seed scheduling, which classifies seeds based on potential memory leak areas in the program and further schedules them, thereby effectively improving the efficiency of discovering memory leak vulnerabilities. Experiments demonstrate that RBZZER outperforms the state-of-the-art fuzzing techniques by finding 52% more program unique crashes than the second-best counterpart. In particular, RBZZER can discover the amount of memory leakage at least 112% more than the other baseline fuzzers. Besides, RBZZER detects memory leaks at an average speed that is 9.10x faster than MemLock.

KEYWORDS: System security; software testing; directed fuzzing; memory leak vulnerability

1 Introduction

Fuzz testing, often referred to as fuzzing, is a widely adopted security testing technique. Due to its high degree of automation and minimal requirement for expert knowledge, fuzzing has been applied across various domains, such as software program testing [1], interface testing [2,3], firmware testing [4,5], vulnerability detection in vehicular networks [6], and vulnerability reproduction [7]. Based on differences in technical characteristics, fuzzing can be categorized into three types: grey-box fuzzing [1], white-box fuzzing [8,9], and black-box fuzzing [10–12]. Among these methods, coverage-based grey-box fuzzing (CGF) achieves the optimal balance between precision and overhead. Unlike black-box fuzzing, which tends to have poor effectiveness, and white-box fuzzing, which is often less efficient, CGF leverages static analysis and lightweight program instrumentation to collect program information. This information is then used to guide the generation of test cases. Owing to its efficiency and scalability, CGF has emerged as the most popular fuzzing technique.

One of the key directions in optimizing CGF tools has been improving program coverage [13–17]. Intuitively, testing more program paths increases the likelihood of discovering unique crashes or vulnerabilities. However, certain vulnerabilities, such as memory leaks, cannot be triggered solely by increasing coverage. Memory leak vulnerabilities are commonly found in programs developed using C/C++. These vulnerabilities cause a gradual depletion of available system memory, negatively impacting overall system performance. In certain application scenarios, such as always-on hosts, memory leaks can lead to memory exhaustion, causing the system to become unresponsive and eventually crash. If such a vulnerability can be exploited by an attacker through carefully crafted scripts, the system becomes exposed to risks such as Denial-of-Service (DoS) attacks or unintended behaviors due to insufficient memory.

Researchers have increasingly recognized the potential risks of memory leak vulnerabilities being exploited to launch DoS attacks [18–20]. However, most tools aimed at detecting memory issues primarily focus on memory corruption vulnerabilities [21–25]. MemLock [26], as the first fuzzing tool specifically designed to detect memory leak vulnerabilities, is capable of automatically identifying such issues without requiring any expert knowledge. MemLock begins by performing static analysis on the target program to identify memory-related statements. These statements are instrumented to monitor memory consumption during program execution. In the fuzzing process, not only are inputs that cover new branches added to the seed queue but inputs that result in increased memory consumption are also retained as "interesting inputs." This mechanism enables MemLock to generate inputs that can make the program's memory consumption exceed the available system memory, thereby triggering memory leak vulnerabilities within an acceptable time frame.

Although the memory monitoring and guidance mechanisms proposed by MemLock are highly effective in detecting memory leak vulnerabilities, the tool still has notable limitations. The guidance mechanism employed by MemLock exhibits a degree of aimlessness in practice. MemLock integrates an AFL-style coverage-guided mechanism with a memory consumption-guided mechanism. While this approach enables the fuzzer to mutate seeds that can trigger program paths with higher memory consumption, the process is entirely random. Specifically, the fuzzer lacks explicit information about which unexplored code regions are more likely to contain memory leak vulnerabilities. As previously mentioned, memory leaks occur when programs fail to properly track allocated memory due to negligence or errors after invoking risk functions e.g., malloc, realloc, and other heap memory operations. Memory leaks can only arise after the invocation of these risk functions. However, prior to identifying memory allocation operations, MemLock treats all code regions as equally important. This means that in the early stages of fuzzing, MemLock is effectively blind to potential memory leak points, leading to significant computational overhead due to a large number of ineffective operations.

We observe that there are three problems that need to be tackled. First, how can we provide a fuzzer with information that enables it to guide the testing towards specific code areas even in the early stages of fuzz testing? Second, when multiple risky functions exist in different code areas of a program, how can we design a reasonable seed scheduling strategy to ensure that the fuzzer does not get stuck in a situation where a large number of seeds in the queue are related to memory leak points that are easier to reach, thus delaying the mutation of seeds related to one or more memory leak points that are associated with other code areas. In other words, how can we address the starvation issue faced by certain memory leak points in a scenario with multiple memory leak points? Third, how can we evaluate the quality of seeds in the queue, measure the potential of seeds to trigger memory leaks, and allocate appropriate resources to them based on their memory leak-triggering potential?

To address the aforementioned three challenges, this paper introduces a solution called RBZZER. Firstly, RBZZER utilizes lightweight static analysis to identify the locations of risky functions in the program,

marking them as potential memory leak points. It then divides and merges code regions based on the similarities of all potential memory leak points in the code areas to create Risk Areas (RA). Secondly, RBZZER borrows the concept from directed fuzz testing techniques [7] to calculate the distance from each basic block in the program to the memory leak points it reaches before execution, inserting these calculations into the basic blocks. Subsequently, seeds are clustered based on RA, and seed selection is performed according to the clustering results. Lastly, RBZZER employs an energy allocation algorithm that measures seed potential based on factors such as seed-based memory consumption and seed distance, to allocate appropriate energy to the seeds. Experimental results indicate that RBZZER outperforms six state-of-the-art tools (i.e., AFL [1], AFLGo [7], AFLfast [27], PerfFuzz [28], QSYM [29] and MemLock [26]), in discovering the memory leak vulnerabilities. RBZZER finds 52% more program unique crashes than the second-best counterpart. In particular, RBZZER can discover the amount of memory leakage at least 112% more than the other baseline fuzzers. Besides, RBZZER detects memory leaks at an average speed that is 9.10× faster than MemLock. Our main contributions are summarized as follows:

- We introduce RBZZER, a fuzzer that effectively identifies memory leak vulnerabilities in programs. The fuzz testing efficiency of this fuzzer is significantly higher than that of the baseline tool MemLock presented in this paper.
- We present a risk areas-based seed scheduling mechanism that effectively enhances the fuzz testing efficiency in multi-target directed fuzz testing scenarios.
- The efficiency of RBZZER was evaluated in multiple popular real-world programs in this study. Experimental results demonstrate that RBZZER outperforms similar fuzz testing techniques in terms of efficiency across various metrics.

The rest of this paper is structured as follows. Section 2 introduces the background and motivation. Section 3 gives the overview and elaborates on the details of RBZZER. Section 4 evaluates our approach. Section 5 introduces the related work before. Section 7 concludes.

2 Motivation

2.1 Problem Description

A memory leak vulnerability means that the software does not sufficiently track and release allocated memory after it has been used, making the memory unavailable for reallocation and reuse [30]. From the description of memory leak vulnerability, it can be observed that the occurrence of memory leak vulnerability is related to memory allocation and release. Fuzzing is an automated software testing technique that detects security vulnerabilities by feeding the target program a large number of unexpected, random, or malformed inputs [31]. Therefore, when fuzzing aims to detect memory leak vulnerabilities, a key challenge lies in generating test cases capable of exercising memory allocation and deallocation-related code paths.

Existing solutions exhibit a degree of *aimlessness*. MemLock, prior to detecting memory allocation operations by the target program, operates according to the logic of AFL, leading to the fuzzer expending considerable time exploring code regions unrelated to memory leak vulnerabilities. For example, the simplified control flow graph of CVE-2019-20023 [32] is shown in Fig. 1a, with BB_4 and BB_15 being the points of memory leakage, where BB means basic block. For brevity in the example, numerous sub-branches between BB_1 and BB_2 have been omitted. From the figure, it can be observed that only 3 basic block paths are capable of triggering the memory leak vulnerability, specifically path (1, 2, 6, 7, 8, 10, 11, 12, 15), (1, 2, 6, 7, 9, 10, 11, 12, 15) and (1, 2, 3, 4). However, existing directing methods explore all possible paths in the program, causing the fuzzer to waste computational resources and slowing down the vulnerability discovery process.

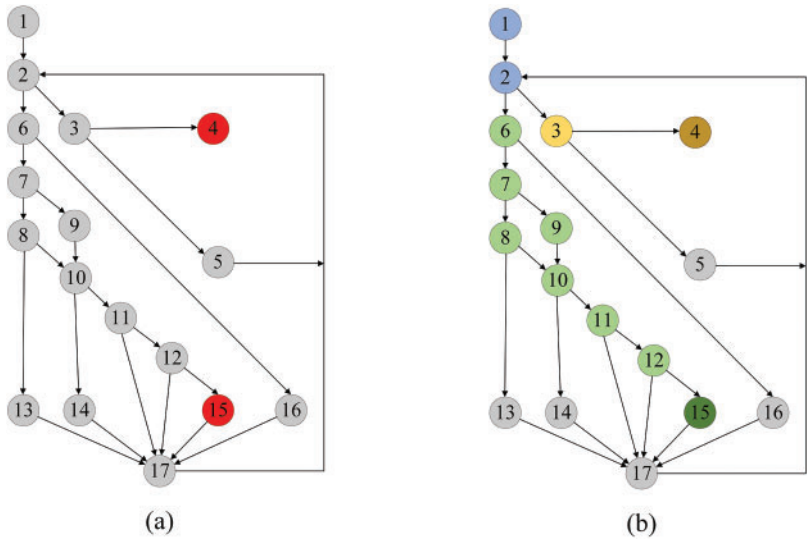

Figure 1: A simplified CFG of CVE-2019-20023, where each node represents a basic block

To validate this defect, we performed a 24-h fuzz testing on three widely-used open-source library components xmllint [33], imginfo [34], and pdftotext [35] using MemLock. Following the testing, we extracted the seed queues generated during the execution of each program and analyzed the seed execution traces using DynamoRIO [36] and Lighthouse [37]. Based on the analysis, we calculated the proportion of seeds in the seed queue that failed to reach the memory operation function area, which we refer to as *unrelated seeds*. The results of this analysis are presented in Fig. 2. As illustrated in the figure, MemLock revealed that 25.52% to 53.86% of the seeds in the queue were unrelated during the exploration of different programs.

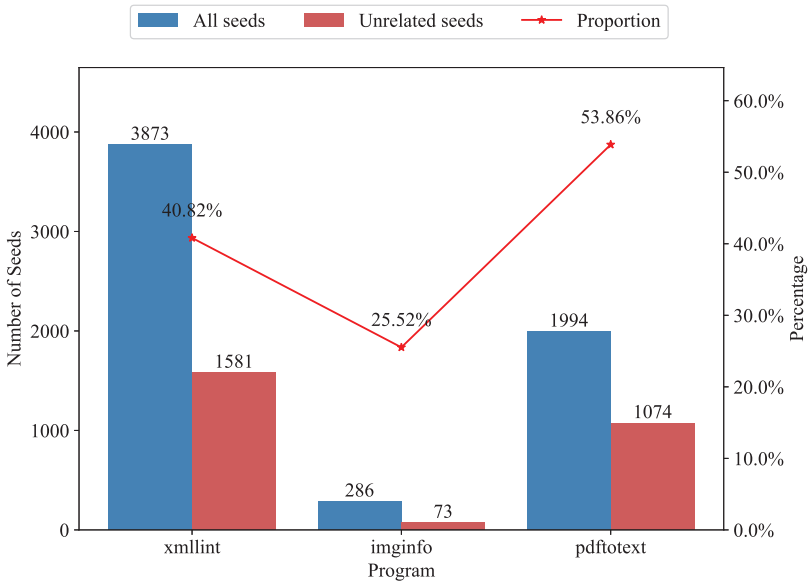

Figure 2: The number of seeds unrelated to PML

2.2 Solution

Intuitively, if the fuzzer could be introduced to produce seeds more likely to reach the Potential Memory Leak Point (PML) in the early stages of fuzz testing, the issue of aimlessness in existing solutions could be effectively addressed. Consequently, RBZZER incorporates the concept of directed fuzz testing. Prior to initiating the fuzz testing process, static analysis is first employed to identify PMLs within the program. Subsequently, the distance from each basic block of the PUT to the PML is calculated. Finally, these calculation results are embedded into the basic blocks, enabling the fuzzer to obtain distance information of the current seed during the fuzz testing process and thereby guide the mutation algorithm. Through these operations, the fuzzer can *anticipate* the location of the PML. In order to ensure clarity in the description, the following definition is provided in this paper:

Definition 1. ***Potential Memory Leak Point (PML)*** *is a special basic block that calls a function which may cause a memory leak, namely a Risk Function. Given the target program PUT, the set of basic blocks constituting the functions bbSet, and the list of risk functions rfList, a Potential Memory Leak Point can be defined as:* $PML = \{bb \in bbSet(PUT) | invoke(bb, rfList)\}$. *Specifically,* $invoke(bb, rfList)$ *represents a basic block that calls a function from the list of risk functions* $rfList$*, where* $rfList = [malloc, calloc, realloc, new]$.

However, simply optimizing MemLock's memory consumption guidance mechanism using directed fuzz testing techniques presents some issues. As shown in Fig. 1a, assuming that after a period of directed fuzz testing, the fuzzer generates a seed that can execute to BB_2 (as mentioned earlier, a large number of irrelevant branches in Fig. 1 have been omitted), and this seed has an execution path of (1, 2, 3, 4). Since BB_4 causes a memory leak, the fuzzer will allocate more energy to this seed and its derived seeds, meaning that more time will be spent exploring code areas related to BB_4, thereby causing starvation of BB_15.

Therefore, RBZZER adopts a seed scheduling strategy based on RA, ingeniously avoiding the starvation problem faced by some PMLs when there are multiple PMLs in the PUT. Specifically, based on the program's Control Flow Graph (CFG), the algorithm traces back from each PML to its predecessor nodes in reverse. The program is then divided into different areas according to the sequence of predecessor nodes for each PML. As shown in Fig. 1b, the yellow area is where BB_4 is located, the green area is where BB_15 is located, and the blue area is the common area for both. After obtaining the basic area information, RBZZER merges areas based on their similarity and then schedules seeds according to the merged results, avoiding the starvation problem faced by leakage points located in different areas.

3 Approach of RBZZER

3.1 Overview

As shown in Fig. 3, the framework of RBZZER consists of two main components: the static analysis component and the fuzz testing component. The static analysis component is responsible for two tasks: 1) generating the control flow graph and the function call graph of the target program, and then calculating the distance from each basic block of the target program to the PML based on these two graphs; 2) identifying the code areas where PMLs are located in the program, and dividing and merging these areas according to the similarity of the code areas to construct Risk Areas. Ultimately, the static analysis component outputs two sets of data: 1) the clustering results of Risk Areas; and 2) the distance calculation results at the basic block level.

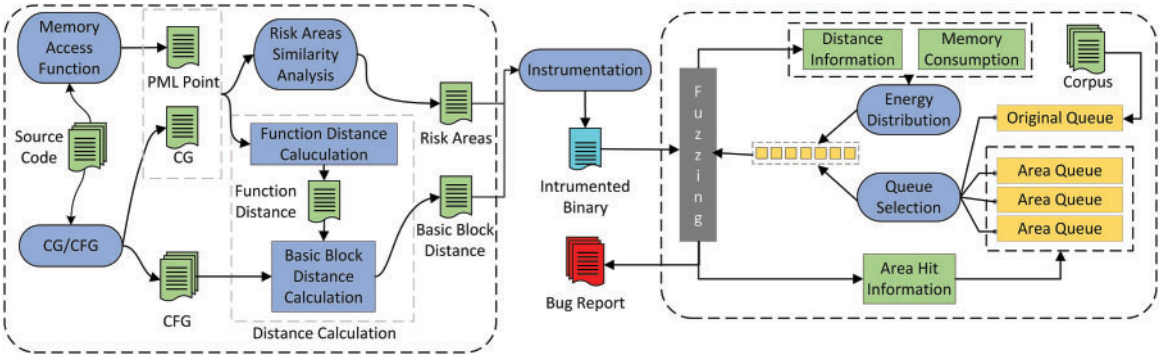

Figure 3: The overall framework of RBZZER

3.1.1 Distance-Guided Mechanism

Code areas in the target program that are unrelated to PML may exist. When the program executes in these areas, memory leakage cannot be triggered no matter what. Allocating excessive energy to explore such irrelevant areas will lead to a waste of performance. Introducing a directed guidance mechanism based on the MemLock memory consumption guidance mechanism can effectively identify the PML-related areas in the program and allocate more energy to seeds that can execute these regions. In the following parts of this paper, such seeds are referred to as PML-related seeds, while the opposite is called PML-unrelated seeds. Note that PML-unrelated seeds are not completely abandoned, but are allocated with lower energy, because mutation of these seeds may still change their execution trajectories, transforming them into PML-related seeds.

Specifically, RBZZER builds upon the classical directed guidance mechanism by considering the context in a probabilistic form. It combines the utilization methods of control flow and data flow information, enabling a more refined assessment of the likelihood of a test case execution path reaching the target point. To effectively leverage data flow and control flow information for more accurate basic block distances, this scheme defines the basic block distance based on deviation probability, optimizing the classical distance algorithm. Subsequently, it dynamically updates the basic block distance by utilizing the data flow characteristics of deviant basic blocks, ultimately obtaining the basic block distance based on deviation probability.

3.1.2 Risk Areas Analysis

To analyze risk areas, it is first necessary to determine the locations of all PMLs in the target program. During the process of traversing the instructions within the basic blocks of the target program for distance calculation, if a function call instruction is encountered, the operands of the instruction are checked. If the current instruction matches the pattern of calling a risky function, its location information (including the name of the basic block where the instruction is located and the function name) is saved locally. In this way, RBZZER obtains the location information of all PMLs in the target program.

Next, it is necessary to construct the risk areas of the target program based on the call graph (CG). First, the scope of the code areas where each PML in the target program is located (PML areas) needs to be determined, that is, the division of code areas. When there are multiple PMLs in the program, a large number of seeds that can execute to the PML areas that are easier to trigger may be generated during the fuzzing process, causing the other PML areas to remain unexecuted for a long time, that is, starvation occurs. However, these code areas may also contain memory leakage vulnerabilities. Therefore, RBZZER needs to determine the scope of each PML region, and then schedule the seeds according to the scope of the areas, so that all PML areas have the opportunity to be executed, thereby avoiding the problem of starvation.

After the division of code regions is completed, the division results need to be merged. The reason for merging code regions is that if there are too many PMLs, the program will be divided into too many PML areas, leading to the degradation and failure of the seed scheduling strategy based on risk regions. To complete the merging of areas, the similarity of each PML region is first calculated. If the similarity between two PML areas is higher than the threshold, they are merged. Finally, a set of PML region clusters identified by unique numbers is obtained. Each cluster is identified by a unique number and contains one or more PML regions. Note that the control flow paths of the PML regions in the same cluster have a higher degree of similarity. If a seed can execute a PML region in the cluster generated during the fuzzing process, there is a higher probability that the seed can execute other PML areas in the same cluster after further mutation. In the following parts of this paper, the PML region cluster is referred to as the Risk Area.

3.1.3 Seed Scheduling Based on Risk Areas

At the beginning of fuzz testing, the fuzzing component adds all seeds to the original queue. During the fuzzing process, if the execution path of a seed enters a risk area, the seed is then added to another queue. This queue has the same identifier as the risk region and contains seeds that can reach the corresponding risk area. This paper refers to this queue as the Area Queue as shown in the figure. After each round of fuzz testing, the fuzzer adaptively selects the seed queue for the next round based on the number of seeds in each risk queue and the number of times the risk queue is selected.

3.2 Distance-Guided Mechanism

In order to guide the fuzzer to mutate seeds that can cover risky areas in the early stages of fuzz testing, RBZZER introduces a directed guidance mechanism. Classic distance algorithms represented by AFLGo can only measure the distance of test cases from the perspective of control flow, neglecting the significant impact of data flow on the program's execution trajectory. WindRanger [38] identified this issue and first proposed the concept of Deviation Basic Block, which refers to the basic blocks in the control flow graph that have unreachable child nodes. Based on this characteristic, the fuzzer would have the capability to use the program's data flow information to guide the fuzz testing process. Although this method incorporates information from the data flow aspect, experiments have shown that the method using deviation basic blocks has limited effectiveness, achieving only about a 40% improvement in efficiency compared to AFLGo. After analysis, we found that WindRanger focused excessively on the deviation basic blocks, ignoring the influence of the program's control flow context on the distance.

To address this issue, we first refer to the definition of the probability of a basic block reaching the target point, and define the basic block distance based on deviation probability, optimizing the classical distance algorithm. Then, by utilizing the data flow characteristics of deviation basic blocks, we dynamically update the basic block distance, ultimately obtaining the basic block distance based on deviation probability. The specific definition is as follows:

Definition 2. ***Basic Block Distance Based on Deviation Probability (BDD)*** *is a basic block distance that integrates deviation probability. Given the basic block distance D_B, the deviation probability P, and the branch condition complexity C, the basic block distance based on deviaton probability BDD is defined as:* $BDD = ln(1 + C * \frac{D_B}{1-P})$.

In the calculation process of BDD, all target basic blocks contained in the program are first collected, and the deviation probabilities of all target basic blocks are set to 0. Then, starting from the target point, the control flow graph is traversed in reverse. For each basic block traversed, its deviation probability is updated to the average of the deviation probabilities of all its successor nodes as the initial deviation probability. If a basic block does not have a successor reachable to the target point, the deviation probability is set to 1.

Considering the impact of the program's data flow, the branch condition complexity of each basic block needs to be calculated. We employ the ratio of the number of executions of branches that cannot reach the target point to the number of executions of branches that can reach the target point as a measure of the complexity of the branch condition.

An example of the calculation process of BDD is shown in Fig. 4. In the control flow graph in the figure, the notation of *X:Y:Z* means that *X* is the basic block distance D_B, *Y* is the deviation probability *P*, and *Z* is the branch condition complexity *C*. For example, node 1:0:1 means that the basic block is directly connected to the target point because *X* equals 1. Due to the basic block being directly connected to the target point, the deviation probability *Y* is 0. The branch complexity under this case is 1 because *both* branches can reach the target point. Therefore, we can figure out the *BDD* of this basic block is ln(2) colored red as shown in the figure. A smaller *BDD* means that the basic block is more likely to reach the target point and vice versa. As shown in the Definition 2, the computational complexity for each variable is $O(1)$, resulting in an overall time complexity of a BDD's computation is $O(1)$. During the overall calculation process, since it is necessary to perform a backward traversal of basic blocks for each PML and then compute BDD for these basic blocks, $O(n^2)$ BDD computations are required. Consequently, the total computational time complexity is $O(n^2)$.

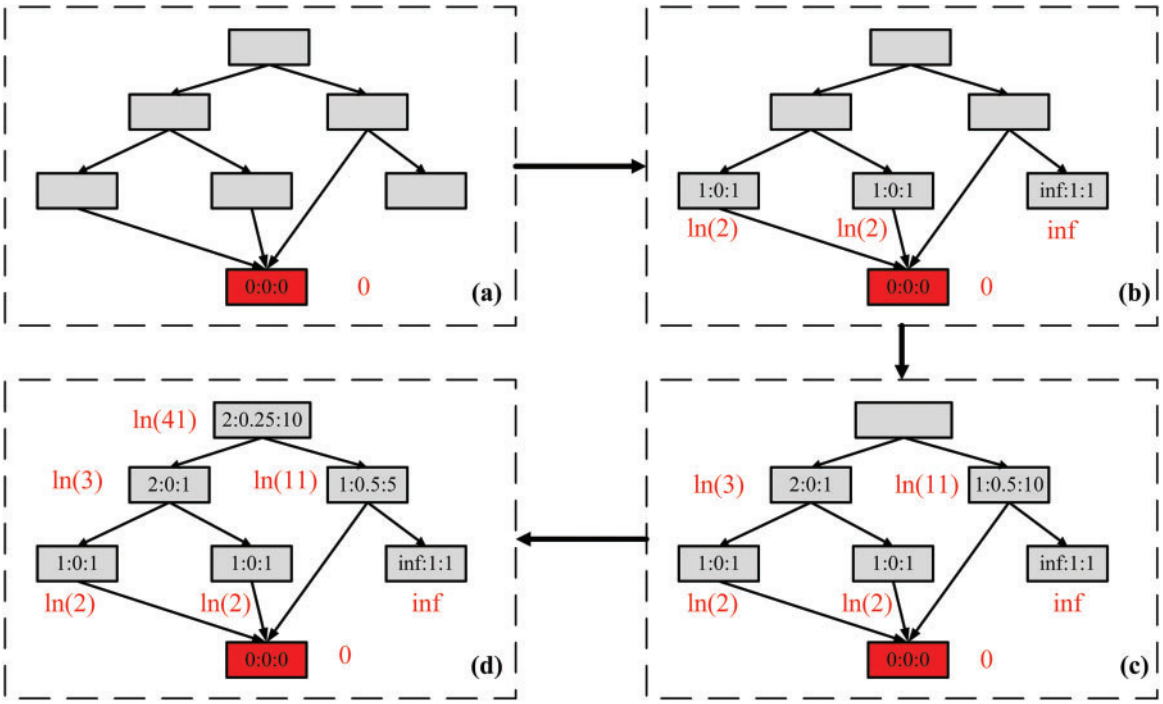

Figure 4: An example of calculation process of BDD: (a) Parameter initialization and computation for the target basic block's BDD; (b)–(d) Subsequent-layer BDD computation performed in reverse order of the CFG

Furthermore, compared to traditional distance metrics, the BDD distance provides more realistic approximations by incorporating the deviation probability between the current basic block and the target basic blocks. Taking basic blocks *1:0.5:5* and *1:0:1* in Fig. 4d as an example, both basic blocks have a uniform distance value of 1 under traditional distance metrics. However, the former basic block contains unreachable a basic block *inf:1:1* in its successor branches, resulting in a significantly lower actual probability of reaching the target block compared to the latter. Consequently, the former basic block should be assigned a larger distance value rather than an equal one. The BDD design accounts for the deviation probability by assigning the former a greater distance value of ln(11) vs. the latter's ln(2) with red font in the figure.

3.3 Risk Areas Analysis

When there are multiple PMLs in a program, a suitable scheduling mechanism is required to ensure no starvation issues mentioned occur. To mitigate the challenges posed by this problem, this paper introduces the concept of a risk area and the definition is as follows:

Definition 3. ***Risk Area*** *is a set composed of a group of basic blocks, where the basic blocks in this set originate from one or more PML. Given a set of n PML, denoted as LP, the set of PML ancestor basic blocks* $preBBset(LP_i)$*, and a threshold ε, the risk area can be defined as:* $RA = \{set(bb) | bb \in \cup_i^n preBBset(LP_i) \wedge bb \notin \cap_i^n preBBset(LP_i), \forall LP_i \in LP, similarity(LP_i) > \varepsilon\}$*, where bb denotes basic block,* $similarity(LP_i)$ *denotes the similarity of* LP_i *with other leakage points in the set LP.*

Definition 3 defines a set of function collections, where PML is necessarily located at some position in the subsequent call chain of the functions in this set. That is, each function node in the risk area has at least one path that can reach PML. To construct the risk areas during static analysis, three sequential steps are required, each implemented by a dedicated algorithm. Firstly, RBZZER identifies all the basic (PMLs) containing the calls to risky functions through static analysis. Secondly, starting from the function containing the PML, the control flow graph and call graph are traversed in reverse to obtain the area corresponding to each PML, which is called the PML area in this paper. Thirdly, the similarity between each PML area is calculated, and they are merged to form the risk area.

Fig. 5 illustrates the step-by-step process of constructing risk areas, demonstrating control flow graph transformations at each stage. Initially, the static analysis yields the program's CFG (Fig. 5a). Algorithm 1 then identifies PMLs within this CFG, producing the annotated graph shown in Fig. 5b, which highlights three PML basic blocks in red. Subsequently, Algorithm 2 constructs vulnerable regions by marking predecessor blocks for each PML basic block (Fig. 5c). Here, areas C_1, C_2, and C_3 (colored in the figure) represent the risk areas corresponding to the three PML basic blocks. Finally, Algorithm 3 merges areas based on area similarity analysis. As visible in Fig. 5c, the risk areas of PML blocks 6 and 7 exhibit high similarity, leading to their merger in the final result (Fig. 5d). The final output shows two risk areas (associated with blocks 1 and 2) indicated by different colors.

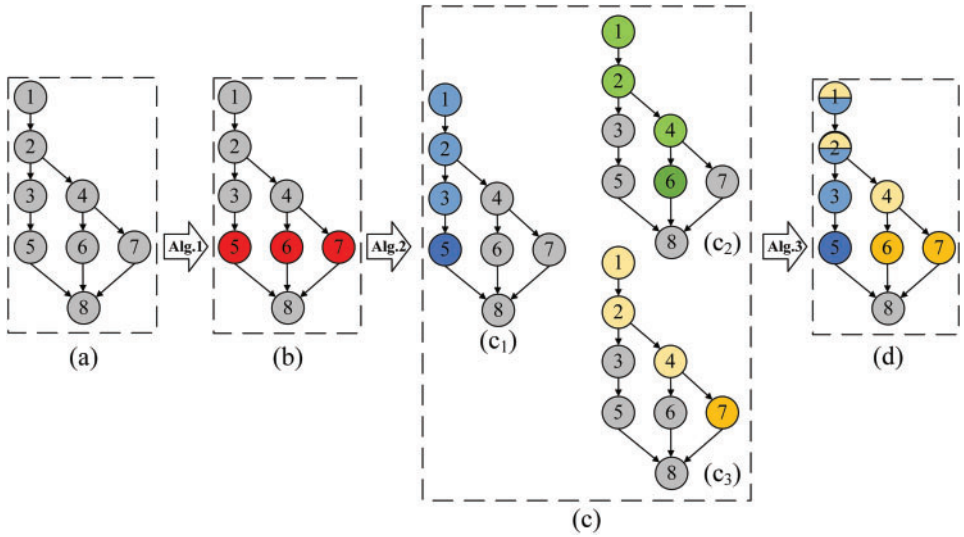

Figure 5: Schematic of the Risk Area construction process

Moreover, the figure illustrates that the risk area guidance mechanism (i.e., the colored areas) narrows the search space for memory leak fuzzing (originally the entire grey area). This reduction enables the fuzzer to discover memory leak vulnerabilities by exploring fewer basic blocks of the program. Here, we can theoretically demonstrate that the set of basic blocks in risk areas of normal programs is smaller than the set of all program basic blocks.

Proof. Let B denote the set of all basic blocks in the program, and $A \subseteq B$ represent the set of basic blocks in risk areas derived through reverse traversal of PML markers. By Definition 1,3, set A excludes all basic blocks positioned after PML markers in the program's forward execution sequence. This implies $|A| < |B|$ unless all leaf-node basic blocks are PML-marked. For normal programs, post-PML execution necessarily contains business logic operations followed by eventual memory releases, thus completing the proof that $A \subset B$. □

To construct the risk area, it is first necessary to determine the location of PML. As shown in Algorithm 1 RBZZER generates the CFG for each function and records the information about internal function calls while traversing the intermediate language functions, basic blocks, and instructions using PASS. It also detects whether each instruction is a call to a memory operation function. If the condition is met, the location of the basic block containing the instruction is recorded. Subsequently, the call graph of the entire program is generated based on the recorded called functions. While the presence of three nested for-loops might indicate $O(n^3)$ complexity, the algorithm's actual behavior constitutes a linear traversal of assembly instructions, resulting in $O(n)$ time complexity for the instruction set.

Algorithm 1: Search PML

Input: The program under test *P*, the risk funciont list *rfList*.
Output: The center point list *cpList*.

```
IC = intermediateCodeGeneration(P, rfList)
cpList = []
for each function F in IC do
    fileName ← getFileName(F)
    lineNum ← getLineNum(F)
    for each basic block BB in F do
        for each instruction Ins in BB do
            Ins ← fileName:lineNum
            cpList.append(Ins)
        end
    end
end
return cpList
```

Algorithm 2: Construct Risk Area

Input: The call graph of PUT *CG*, the center point list *cpList*.
Output: The risk areas consists of basic blocks *riskArea*.

```
AWM ← initAWM(CG)
riskArea = [], Q = []
Q ← initWithEmptyQueue()
```

(Continued)

Algorithm 2 (continued)

```
4 for each center point CP in cpList do
5     AWM[CP].weight_nodes.insert(CP)
6     Q.push(CP)
7     while Q not empty do
8         curNode ← Q.front()
9         Visited.append(curNode)
10        for each previous node PN in curNode.prevNodes do
11            riskArea[CP].insert(CP)
12            if PN not in Visited then
13                Q.push(PN)
14            end
15        end
16    end
17 end
18 return riskArea
```

Upon obtaining the CFG, CG, and the list of center points, area division can be carried out. To proceed with the subsequent tasks, we have designed a directed graph data structure, termed the Area Weighted Map (AWM). Each node in this graph structure is capable of accessing all its parent and child nodes and possesses a set of weight nodes. Unlike the conventional concept of weight, the weight node set is a collection composed of function node names, with each element representing the name of a center point of a particular area. Initially, the weight node set of each node is empty. After reading the CG file and initializing the AWM, the weight of the center point node is set to its function name. Subsequently, starting from each center point node, all its parent nodes are visited in a reverse manner. For each visited node, its weight node set is inserted into the weight node set of each of its parent nodes, and its parent nodes are added to the visiting queue. This operation is repeated until, in the end, each node in the AWM has 1 or more weight nodes. At this juncture, the scope of each center point is initialized, which is a set identified by the center point name. The AWM is traversed, and if a node is found to carry a center point name as a weight node, that node is incorporated into the center point's scope. It should be noted that this operation may result in some nodes being included in multiple scopes. Lastly, Algorithm 2 returns a mapping that encompasses the basic block information within the scope of each center point. The algorithm's structure consists of two nested for-loops along with a termination-checking while-loop. As its fundamental operation performs a basic-block-level backward traversal for each PML, the algorithm exhibits $O(n^2)$ time complexity.

After determining the scope of each area center, it is necessary to continue merging the ranges of various areas in order to achieve optimal efficiency. Intuitively, if two areas A and B have a large overlapping region, there is a higher probability that a seed that can reach area A can also reach B through mutation.

Trajectory similarity has a wide range of application scenarios, such as recommendation algorithms based on similar travel routes, clustering of vehicle driving habits, and infectious disease prevention and control based on human movement trajectories. Trajectory distance is a measure used in trajectory similarity algorithms to gauge the similarity between trajectories; the greater the trajectory distance, the greater the difference between the two trajectories, and vice versa. Since areas are divided based on the CFG, an area can be regarded as a set of program execution trajectories. Therefore, trajectory similarity can be used to assess the similarity between areas. The Longest Common Subsequence (LCSS) algorithm is used to calculate the

longest common subsequence between two sequences. Let sequence A be of length n and sequence B be of length m, the method for calculating the length of their longest common subsequence is shown in Eq. (1), where γ is a member similarity threshold and the default value is 0.8, $t \in (0, n]$, $i \in (0, m]$.

$$LCSS(A,B) = \begin{cases} 0, & if\ A = \varnothing\ or\ B = \varnothing \\ 1 + LCSS(a_{t-1}, b_{i-1}), & if\ dist(a_t, b_i) < \gamma \\ max(LCSS(a_{t-1}, b_i),\ LCSS(a_t, b_{i-1})), & otherwise \end{cases} \tag{1}$$

Based on the length of the longest common subsequence, the distance between area A and area B can be determined, with the specific calculation method shown in Eq. (2). The longer the longest common subsequence, the higher the similarity between the two areas.

$$D_{LCSS} = 1 - \frac{LCSS(A,B)}{min(|A|, |B|)} \tag{2}$$

For now, the task of merging similar areas can be accomplished. Specifically, as shown in Algorithm 3, the similarity between each pair of areas is calculated using the LCSS algorithm, and the results are recorded in a similarity matrix. Then, the values in the similarity matrix are examined, and areas with similarity values above the threshold are merged to obtain the risk areas. The algorithm requires pairwise comparisons of risk areas. Considering that the LCSS algorithm has a time complexity of $O(m \times n)$, where m and n represent sequence lengths, the overall time complexity ranges between $O(min(m,n)^2)$ at best and $O(max(m,n)^2)$ at worst.

Algorithm 3: Merging Risk Area

Input: The risk areas *riskArea*, the threshold of similarity θ.
Output: The merged area M.

```
Matrix = [][]
for each area A in riskArea do
    for each area A' in riskArea do
        if A == A' then
            Matrix[A][A'] = 100
        end
        similarity = D_LCSS(A, A')
        Matrix[A][A'] = similarity
    end
end
M = merge(M, riskArea, θ)
return M
```

In the end, several risk areas will be obtained, each of which contains one or more center points, and these center points represent the potential areas of memory leak vulnerabilities. RBZZER assigns a unique ID to each area. During the instrumentation process, based on the results of area division and merging, statements that write the ID of the area where the basic block is located to the shared memory are inserted into the basic block. In this way, RBZZER can dynamically monitor the execution trajectory of the seed during its execution, and further carry out seed scheduling.

3.4 Seed Scheduling Based on Risk Areas

After the completion of the area merging operation, if there are multiple areas, the areas located at deeper levels of code logic may face the problem of starvation. This is because, if the fuzzer simply uses the seed's memory consumption and the distance of the seed as the guiding mechanism, it will preferentially select seeds with smaller distances and shorter paths, thereby falling into a local optimal solution. To address this issue, this paper proposes an area-based seed scheduling mechanism to avoid the problem of starvation. The mechanism first acquires the area information covered during the execution of the seed and then carries out seed scheduling based on this information.

3.5 Area Information Acquisition

During the execution of PUT, if a seed executes a basic block within a certain area, the instrumentation code inserted into that basic block will write the area number where the basic block is located in the shared memory. After execution, the fuzzer reads the number from the shared memory and adds this mark to the seed. Note that a seed may carry multiple area marks, which occurs when two areas have a low similarity and have not been merged. When executing in the intersection region of the two areas, the marks of both areas are written into the shared memory. In practice, to distinguish the area marks carried by seeds with lower memory overhead, RBZZER marks seeds by shifting 1 to the left of the area mark bit. This method allows RBZZER to distinguish up to 8 areas with a single byte (in practice, there will not be many areas).

3.6 Speed Scheduling

The seed scheduling mechanism aims to prevent seed starvation issues. During the scheduling process, seeds reaching different risk areas are first organized into respective seed queues based on their areas. Subsequently, each queue is scored and the highest-scoring queue is selected for fuzzing. Specifically, after the fuzzer acquires the area mark of a seed, it parses the mark and adds the seed to the corresponding queue. For instance, if a seed carries the mark 40, the fuzzer will parse it as 5 and 3 ($(40)_{10} = (10100)_2$), and then add the seed to the queues numbered 5 and 3. At the end of each fuzzing loop, the fuzzer increments the execution count of the current queue by 1, and then selects a new seed queue for fuzz testing. After each round of fuzz testing, the fuzzer calculates the scores of all queues using Eq. (3), and then selects the queue with the highest score for use in the next round of fuzz testing. In the following equation:

$$P = c[j] * w[j] * \sqrt{\frac{2 * V[j] * log(N)}{N[j]}} \tag{3}$$

In the equation, P is the score of the current queue j. $c[j]$ is the validity coefficient of queue j. When it equals 1, it signifies the presence of a seed within the queue, whereas a value of 0 indicates the contrary. $w[j]$ is the value decay coefficient of queue j, which is used to evaluate the efficiency of discovering new execution paths within the area after all PMLs corresponding to the queue have been hit. When it equals 1, it means the center is not covered, whereas a value of $\frac{G[j]}{R[j]}$ on the contrary. $G[j]$ is the number of seeds found new paths and $R[j]$ is the number of PMLs in the area. If the efficiency is low, the queue will be selected with a lower probability. $V[j]$ is the execution speed of queue j. N is the number of times for queue is selected. $N[j]$ indicates the number of times queue j has been selected. In short, the fuzzer will prioritize selecting the queue with at least one seed that has never been selected before. If all queues have been selected, it will prioritize the queue with high path discovery efficiency, fast execution speed, and fewer selections. During the seed scheduling process, each seed queue requires score computation for selection. As shown in the Eq. (3), the

computational complexity for each variable is $O(1)$, resulting in an overall equation complexity of $O(1)$. Consequently, the total time complexity for seed scheduling is $O(n)$.

4 Evaluation

In this section, with the implemented prototype of RBZZER, we conducted experiments on different applications to answer the following research questions:

- RQ1 How effective is the proposed directed guidance mechanism based on deviation probability?
- RQ2 Does the proposed memory leak guidance mechanism address the issue of aimlessness and area starvation?
- RQ3 How capable is RBZZER in memory consumption crash detection?
- RQ4 How effective is the proposed method in detecting memory leak vulnerabilities?

4.1 Experimental Setup

To evaluate the efficacy of RBZZER, we conducted a comparative study with six cutting-edge fuzzers: AFL [1], AFLGo [7], AFLfast [27], PerfFuzz [28], QSYM [29], and MemLock [26]. The selection of these baseline fuzzers was guided by several factors. AFL and AFLGo are well-established as coverage-based and directed grey-box fuzzers, respectively, and are commonly used as baselines in most research. AFLfast represents an enhanced version of AFL, featuring an improved power schedule. PerfFuzz focuses on stressing time complexity issues within programs, whereas RBZZER is designed to identify space complexity issues. QSYM is a widely recognized symbolic execution-assisted fuzzer. Lastly, MemLock is the best fuzzer that detects memory leak vulnerabilities. In summary, we chose a diverse range of representative state-of-the-art fuzzers as our baselines, which are extensively employed in real-world vulnerability detection efforts.

We select evaluation benchmarks considering several factors, e.g., popularity, frequency of being tested, development activeness, and functional diversity. We reference the fuzzing testing guidance standard of Unifuzz [39]. Finally, 6 commonly used real-world programs from popular open-source libraries were selected as the target programs for testing, which all have disclosed memory leak vulnerabilities as shown in Table 1. Among them, xmllint, pdftotext, and readelf are popular document processing tools, imginfo is a popular image processing tool, swftophp is a popular Flash processing tool, and mp42hls is a popular audio processing tool. These programs have also been widely tested by existing state-of-the-art grey-box fuzzers [40–42].

Table 1: The basic information of 6 programs under test

No.	Library	Version	Program	Format	Lines
1	libxml2 [33]	2.11.7	xmllint	XML	71,448
2	jasper [34]	2.0	imginfo	JPEG	82,745
3	Xpdf4 [35]	4.00	pdftotext	PDF	147,115
4	binutils [43]	2.29	readelf	ELF	183,927
5	libming [44]	0.4.8	swftophp	FLASH	300,985
6	bento4 [45]	1.5.1	mp42hls	MP4	173,468

To evaluate the optimization efficiency brought about by the directed guidance mechanism based on deviation probability, this experiment referred to the Time To Exposure (TTE) as the primary assessment indicator, which is widely adopted in the field of directed fuzz testing [7]. TTE represents the time elapsed

from the commencement of fuzz testing to the generation of a test case by the fuzzer that can trigger the target point. The smaller this value is, the less time it takes to trigger the target point, indicating a better effect of the directed guidance mechanism. Each program was fuzzed 5 times, with each run lasting 24 h. To evaluate the fuzzers, we conducted experiments on a machine equipped with an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40 GHz, running Ubuntu 20.04.6 LTS.

4.2 RQ1: Effect of Directed Guidance

To verify the optimization efficiency of the directed guidance mechanism based on deviation probability, this experiment referenced the experimental setup of UniFuzz and selected 8 locations from 6 target programs as shown in Table 2 as target points. Directed fuzz testing was conducted on these target points using RBZZER and other 6 baseline tools.

As shown in Table 2, in the fuzz testing targeting each of the target points, RBZZER was on average 3.07x, 1.77x, 3.12x, 3.52x, 3.72x, and 2.79x respectively faster than the state-of-the-art fuzzers AFL, AFLGo, AFLfast, PerfFuzz, QSYM and Memlock. This is because the proposed approach corrected the errors in the distance definition within the baseline tool by employing deviation probability and dynamically adjusted the distance of test cases in conjunction with data flow information, enabling the fuzzer to more accurately assess the distance between test cases and target points. With the assistance of directed fuzzing techniques, AFLGo and RBZZER demonstrate a significant acceleration effect compared to the other 5 fuzzers. This indicates that the approach presented in this paper can stably and effectively enhance the efficiency of directed fuzz testing and is capable of triggering the target points at a faster rate.

Table 2: The time of 7 fuzzers consumed to expose vulnerabilities on 8 target sites of programs for directed fuzzing test. TO denotes that a tool reaches the time limit (timeout) before triggering a vulnerability

No.	Fuzzer		xmllint		imginfo	pdftotext		readelf	swftophp	m42hls	Avg. factor
			valid.c: 2637	xmlint.c: 3108	jpc_tsfb.c: 97	gmem.cc: 140	gmem.cc: 187	readelf.c: 21524	parse.c: 101	Mp42Hls.cc: 1217	
1	AFL	Time	523	4933	985	TO	24338	2875	1488	10846	**3.07x**
		Factor	6.79	2.88	2.13	–	2.32	3.39	2.35	1.61	
2	AFLGo	Time	188	2947	553	33977	19115	1830	958	11856	**1.77x**
		Factor	2.44	1.72	1.20	1.54	1.82	2.16	1.52	1.76	
3	AFLfast	Time	575	4689	907	TO	28747	2539	1573	9855	**3.12x**
		Factor	7.47	2.73	1.96	–	2.74	3.00	2.49	1.46	
4	PerfFuzz	Time	673	5277	1327	TO	29643	3438	1627	12837	**3.52x**
		Factor	8.74	3.08	2.87	–	2.83	4.05	2.57	1.90	
5	QSYM	Time	680	5762	1010	TO	24870	3166	1403	13285	**3.72x**
		Factor	8.83	3.36	2.19	–	2.37	3.73	2.22	1.97	
6	MemLock	Time	403	5032	923	69486	27824	2245	1310	10921	**2.79x**
		Factor	5.23	2.93	2.00	3.16	2.65	2.65	2.07	1.62	
7	**RBZZER**	Time	**77**	**1715**	**462**	**22019**	**10482**	**848**	**632**	**6749**	**1.00x**

4.3 RQ2: Evaluation of Aimlessness and Starvation in RA

To verify the effectiveness of the proposed approach in addressing the issue of aimlessness, the experiment utilized RBZZER to conduct tests on 6 target programs and extracted the seed files generated by the fuzzer. Subsequently, DynamoRIO [36] and Lighthouse [37] were employed for seed execution trace analysis, to statistically determine the proportion of seeds in the queue whose execution traces deviated from the code regions leading to the target points.

The experimental results are shown in Table 3. During the vulnerability mining process for each target program using RBZZER, 4653, 305, 2894, 1761, 732, and 3496 test cases were generated, respectively, and all test cases were relevant seeds, with a relevance rate of 100%. This indicates that the approach presented in this chapter can effectively filter out test cases related to memory leak vulnerabilities during the fuzz testing process and discard other redundant test cases.

Table 3: The number of seeds and PML related seeds after fuzzing campaign of RBZZER on 6 programs

No.	Program	Total seeds	Related seeds	Proportion
1	xmllint	4653	4653	100%
2	imginfo	305	305	100%
3	pdftotext	2894	2894	100%
4	readelf	1761	1761	100%
5	swftophp	732	732	100%
6	mp42hls	3496	3496	100%

To verify whether RBZZER can alleviate the starvation problem in risk areas and ensure that all risk areas of the programs are explored, we constructed risk areas for six target programs and conducted tests. The results of the risk area construction are shown in Table 4. For the six target programs, we constructed 22 risk areas, specifically 2, 4, 5, 3, 4, and 4 risk areas for each program, respectively.

Table 4: The number of risk areas of 6 programs

No.	Library	Version	Program	Risk area
1	libxml2	2.11.7	xmllint	2
2	jasper	2.0	imginfo	4
3	Xpdf4	4.00	pdftotext	5
4	binutils	2.29	readelf	3
5	libming	0.4.8	swftophp	4
6	bento4	1.5.1	mp42hls	4

Subsequently, fuzz testing was conducted on the 6 programs under test, and the seed queues corresponding to each risk area were obtained, with the number of seeds in each queue being tallied. Concurrently, DynamoRIO and Lighthouse were utilized to perform execution trace analysis on the seeds in each risk queue, calculating the proportion of seeds whose execution traces deviated from the risk areas. The experimental results are depicted in Fig. 6. During the fuzz testing of the 6 programs under test using RBZZER, no empty risk area seed queues were identified. The risk area queue with the fewest seeds was Queue 0 of the risk area for pdftotext, which contained only 30 seeds. The execution traces of the seeds within each risk area queue were all located within the respective risk areas, meaning that the proportion of relevant seeds in each risk area queue was 100%.

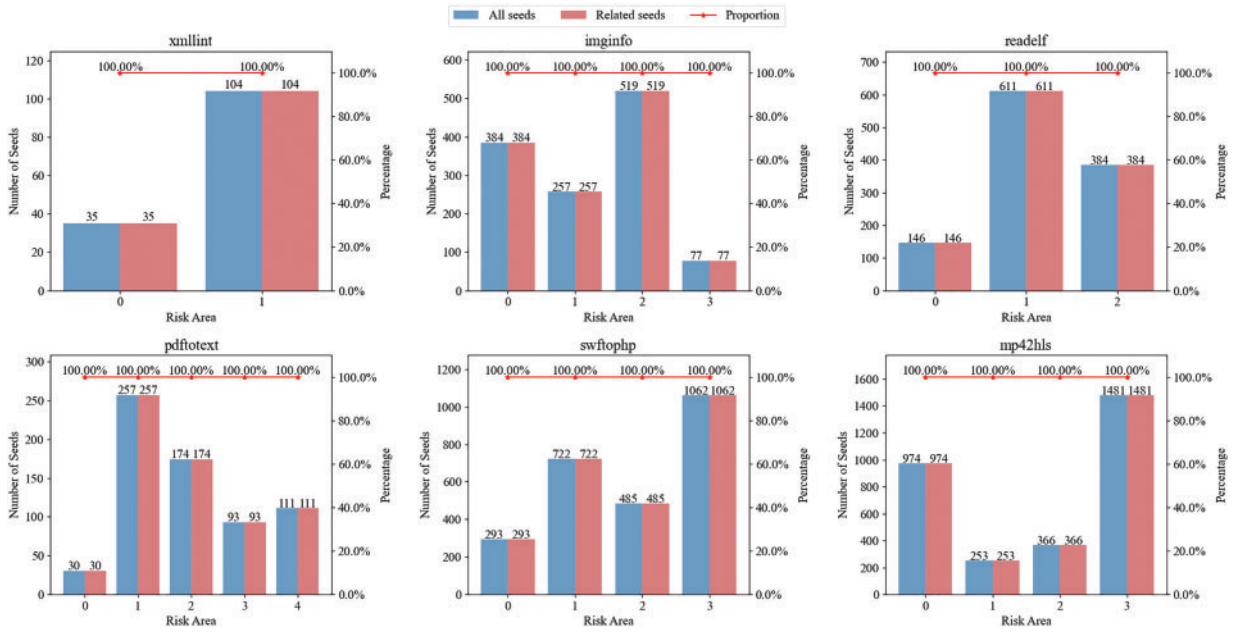

Figure 6: The number of seeds and risk area related seeds after fuzzing campaign of RBZZER on 6 test programs

In summary, RBZZER is capable of effectively constructing risk areas and clustering the seeds generated by the fuzzer based on these risk areas. Moreover, all risk areas were explored, with no risk areas suffering from starvation. This outcome indicates that RBZZER can effectively address the problem of starvation.

4.4 RQ3: Evaluation of Unique Crashes

To assess the effectiveness of fuzzers, one effective metric is the count of unique crashes identified by various fuzzers. Generally, a higher number of unique crashes suggests a greater likelihood of uncovering additional unique vulnerabilities.

As shown in Table 5, the number of crashes found by 7 different fuzzers within 24 h in the 6 target programs. It is worth noting that the crashes are all related to memory leak bugs. It can be observed from the table that RBZZER discovered more crashes in all 6 target programs, with an improvement of 127%, 109%, 111%, 154%, 102%, and 52%, respectively, compared to state-of-the-art fuzzers AFL, AFLGo, AFLfast, PerfFuzz, QSYM and MemLock. Among them, AFLGo performed surprisingly well, primarily because we specified the locations of memory leak risk areas in this experiment. This demonstrates that targeted guidance can effectively enable the fuzzer to trigger specific bugs. Moreover, we also conduct a statistical test for the results. We apply the *Mann-Whitney U*-test [46] with a significance level of 0.05 to check the statistical significance differences of experimental results. The *p-val* statistic measures the probability that RBZZER outperforms another fuzzer. A smaller statistical significance difference indicates a more significant difference between MemLock and the competitor. Thus, we can conclude that RBZZER significantly outperforms the other 6 state-of-the-art fuzzers in benchmark programs.

Table 5: The number of unique crashes related to memory leakage after 24 h run of 7 fuzzers on 6 test programs

No.	Fuzzer		xmllint	imginfo	pdftotext	readelf	swftophp	mp42hls	Total (Improve.)
1	AFL	Crashes	58	25	14	5	22	25	**149 (+127%)**
		p-val	<0.01	<0.01	<0.01	0.04	<0.01	<0.01	
2	AFLGo	Crashes	62	**28**	19	5	23	25	**162 (+109%)**
		p-val	0.01	0.02	<0.01	<0.01	<0.01	<0.01	
3	AFLfast	Crashes	66	26	16	5	**24**	23	**160 (+111%)**
		p-val	<0.01	<0.01	<0.01	0.01	<0.01	<0.01	
4	PerfFuzz	Crashes	64	5	18	3	23	20	**133 (+154%)**
		p-val	<0.01	0.01	<0.01	<0.01	<0.01	<0.01	
5	QSYM	Crashes	74	21	16	6	**24**	26	**167 (+102%)**
		p-val	<0.01	<0.01	0.01	0.02	<0.01	<0.01	
6	MemLock	Crashes	96	22	21	13	17	53	**222 (+52%)**
		p-val	0.01	<0.01	<0.01	<0.01	0.01	<0.01	
7	**RBZZER**	Crashes	**135**	25	**37**	**49**	23	**69**	**338 (+0%)**

4.5 RQ4: Evaluation of Memory Leakage

To verify the advantages of RBZZER in detecting memory leaks, this experiment employed the approach presented in this paper and other 6 fuzzers as the baseline tool to conduct fuzz testing on 6 target programs for 24 h repeated 5 times, recording the maximum memory leakage detected by both tools.

Table 6 presents the specific number of memory leaks detected in the experiments. It can be observed from the table that the maximum number of memory leaks detected by RBZZER in each program under test with an improvement of 2067%, 1083%, 1391%, 1794%, 1994% and 112%, respectively, compared to state-of-the-art fuzzers AFL, AFLGo, AFLfast, PerfFuzz, QSYM and MemLock. The *p-val* indicates that RBZZER significantly outperforms the other 6 fuzzers. MemLock detected more memory leakage on mp42hls, the primary reason for this phenomenon is that the target program under test contains a relatively small number of memory operation function calls, and the fuzzer has designated only a few target points, thus no starvation issue has arisen. Since the stub code volume of RBZZER is larger than that of the baseline tool MemLock, in the absence of a starvation problem, this scheme requires more time to execute the stub code. Consequently, within the same time frame, the number of memory leaks detected by this tool is fewer.

Table 6: The amount of memory leakage after 24 h run of RBZZER and MemLock on 6 test programs

No.	Fuzzer		xmllint	imginfo	pdftotext	readelf	swftophp	mp42hls	Total (Improve.)
1	AFL	Leakage	97.29 KB	38.72 KB	3.88 MB	4.48 MB	47.39 KB	15.13 KB	**8.76 MB (+2067%)**
		p-val	<0.01	0.01	<0.01	<0.01	0.04	<0.01	
2	AFLGo	Leakage	205.73 KB	33.94 KB	3.02 MB	9.98 MB	1.24 MB	1.19 MB	**16.04 MB (+1083%)**
		p-val	<0.01	<0.01	<0.01	<0.01	<0.01	0.01	
3	AFLfast	Leakage	329.48 KB	35.80 KB	4.47 MB	7.48 MB	31.58 KB	90.23 KB	**12.72 MB (+1391%)**
		p-val	0.02	<0.01	<0.01	0.01	<0.01	<0.01	
4	PerfFuzz	Leakage	142.42 KB	15.92 KB	5.82 MB	3.28 MB	530.35 KB	9.84 KB	**10.01 MB (+1794%)**
		p-val	<0.01	0.01	0.02	<0.01	<0.01	<0.01	
5	QSYM	Leakage	193.26 KB	28.44 KB	3.29 MB	4.15 MB	832.93 KB	1.38 MB	**9.06 MB (+1994%)**
		p-val	<0.01	0.02	<0.01	<0.01	<0.01	<0.01	
6	MemLock	Leakage	4.19 MB	477.23 KB	33.16 MB	45.49 MB	1.62 MB	**2.64 MB**	**89.67 MB (+112%)**
		p-val	<0.01	<0.01	0.03	<0.01	<0.01	<0.01	
7	**RBZZER**	Leakage	**10.73 MB**	**46.90 MB**	**34.55 MB**	**72.83 MB**	**18.10 MB**	2.18 MB	**189.74 MB (+0%)**

Moreover, given that RBZZER and MemLock detected significantly more memory leaks compared to other fuzzers, we proceeded to compare the memory leak detection speeds of MemLock and RBZZER, and we recorded the time taken by both to detect the same amount of memory leakage. The experimental

results are shown in Fig. 7. Compared with MemLock, RBZZER achieved a maximum speedup of 38.7x and a minimum speedup of 1.12x in detecting memory leaks during the fuzz testing of each program under test, with an average speedup of 9.10x. This is because, with the assistance of the directed guidance mechanism, the fuzzer can filter out a large number of irrelevant seeds in the early stage of fuzz testing, and use the saved time to discover potential memory leak vulnerabilities in the program.

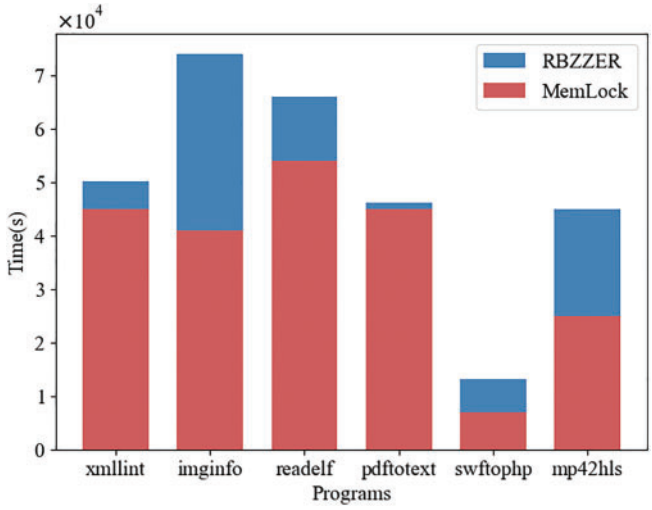

Figure 7: The time consumed to detect the same amount of memory leaks of RBZZER and MemLock

4.6 Case Study

Taking the memory leak in *xmlPatternCompile* function discovered during our experiments with *xmllint* as an example, we analyze the test process outputs of different fuzzers to better understand how RBZZER differs from other approaches. We also utilized DynamoRIO and Lighthouse to examine seeds' traces, counting the number of seeds that reached the risk area and the memory lekage triggered within the 24-h testing period at each time checkpoint, with the results illustrated in Fig. 8.

From the results, we observe that RBZZER was the fastest to generate seeds reaching the risk area, followed by AFLGo. In contrast, MemLock and other fuzzers lacking directed guidance mechanisms generally underperformed, taking approximately 9 h longer than RBZZER to produce seeds covering the target regions. This demonstrates the effectiveness of directed guidance mechanisms. Furthermore, since RBZZER also retains a memory consumption guidance mechanism, it consistently triggered larger memory leaks compared to AFL, AFLGo, AFLfast and other approaches lacking this capability. Overall, the testing process data presented in the figure validates the effectiveness of RBZZER's design.

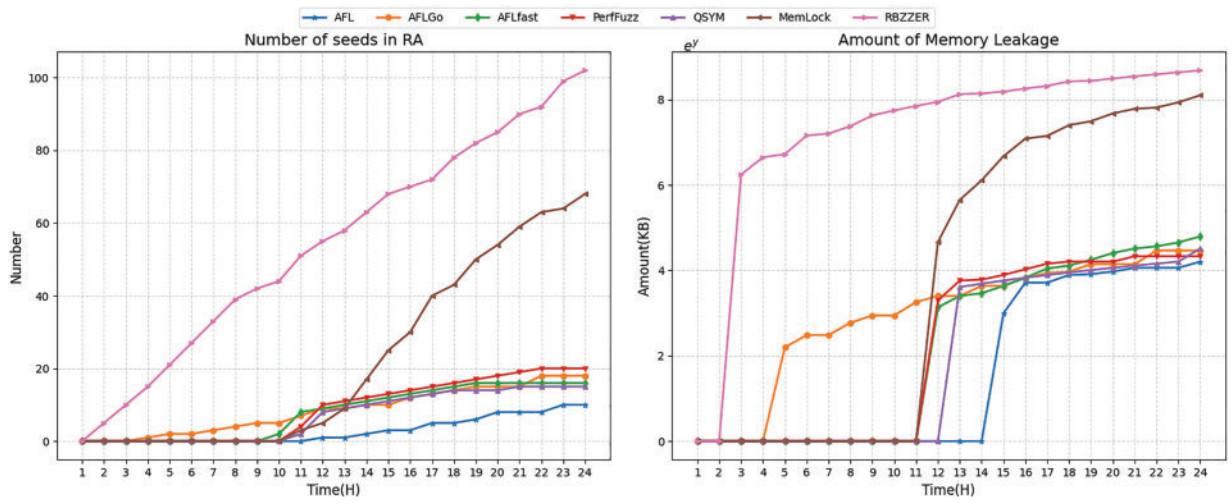

Figure 8: The number of seeds generated from and amount of memory leakage detected by 7 fuzzers in 24 h on an risk area in xmllint

5 Related Work

5.1 Memory Vulnerability Detection Technology

Static memory vulnerability detection techniques conduct static analysis on the source code, compilation of intermediate data, and executable files of the target program. Smoke [47] proposed a method for detecting memory leak vulnerabilities in large-scale code projects, enhancing the scalability and detection accuracy of memory leak vulnerability detection techniques. PCA [48] proposed a solution for detecting memory leak vulnerabilities using static analysis techniques based on data flow to capture abnormal data flows with lower overhead, thereby identifying memory vulnerabilities in programs. MVD+ [49] is a memory vulnerability detection method based on deep learning. This method adopts a hierarchical representation learning strategy to learn the syntactic and semantic features of vulnerable code, which has improved the accuracy of detecting memory-related vulnerabilities and reduced the probabilities of false positives and false negatives.

Dynamic memory vulnerability detection techniques monitor the memory allocation, deallocation, and access behaviors during the execution of a program. Valgrind [50] simulates the execution of a program, monitoring dynamic memory allocation and deallocation in real-time, as well as detecting potential memory errors during runtime. Fuzz testing is a popular dynamic vulnerability mining technique currently in use. Scholars have recognized the efficiency of fuzz testing in memory vulnerability mining and have proposed research plans [21,24,25]. However, the aforementioned studies have focused on discovering memory corruption vulnerabilities and have not paid attention to memory leak vulnerabilities. Unlike memory corruption vulnerabilities, memory leak vulnerabilities do not immediately cause errors or trigger other abnormal conditions when triggered; only when the program repeatedly triggers such vulnerabilities will abnormal situations truly occur. Therefore, detecting memory leak vulnerabilities is more time-consuming than detecting memory corruption vulnerabilities. MemLock [26], as a popular fuzz testing tool specifically designed for detecting memory leak vulnerabilities in programs, can automatically discover memory leak vulnerabilities. This tool first performs static analysis on the program to identify statements related to memory consumption and instruments these statements to monitor the memory consumption during the program's execution. Although the memory monitoring and guidance mechanism proposed by MemLock is

very effective in detecting memory leak vulnerabilities, it still has issues: The guidance mechanism used by MemLock is blind in practice.

5.2 Directed Fuzzing Techniques

Directed fuzz testing techniques guide the fuzzer to generate test cases that can reach specific target locations in the program more efficiently by calculating the distance between test cases and the program's target points. Table 7 presents a summary and comparison of strategies employed by relevant directed fuzzers. These approaches address the performance waste issues faced by grey-box fuzz testing techniques in scenarios such as vulnerability reproduction and patch verification, as exemplified by AFLGo [7]. Hawkeye [51] has conducted a comprehensive optimization of AFLGo, redefining the distance metric and considering and analyzing issues such as function reachability, pointer indirect calls, and seed energy allocation. WindRanger [38] pointed out that the distance values calculated in Hawkeye and AFLGo do not truly represent the difficulty of test cases reaching the target points, as they lack the utilization of the difficulty in satisfying the constraints in the program execution paths. Based on this, WindRanger proposed a solution based on deviation basic blocks. In response to the path explosion problem in directed fuzz testing, BEACON [52] proposed a lightweight infeasible path pruning method based on static analysis, which to some extent alleviates the troubles brought by path explosion to directed fuzz testing. VDFuzz [53] proposes a vulnerability-oriented directed strategy that integrates static analysis to pinpoint high-risk code regions and dynamic fuzzing to guide test cases toward these vulnerable paths. This hybrid approach enhances bug detection precision and speed in binary programs, outperforming traditional fuzzing methods. SFDS [54] proposes a directional seed generation framework to accelerate fuzzing that enables Human-In-The-Loop (HITL) directed fuzzing where the human assumes a more active role in the creation of seeds that can penetrate and assess desired locations of the program under test.

Table 7: The strategy of relevant directed fuzzers

Strategy	AFLGo [7]	Beacon [52]	Hawkeye [51]	SFDS [54]	VDFuzz [53]	WindRanger [38]
Distance	✓	✓	✓	×	×	✓
Path prune	×	✓	×	×	×	×
Weighted coverage	×	×	✓	x	✓	x
Human-In-The-Loop	×	×	×	✓	×	×

6 Limitations and Future Work

6.1 Risk Assessment Metric for PML/Risk Areas

The directed fuzzing mechanism proposed in this work incorporates automated identification of PML by scanning memory operation function calls and designating them as target locations, significantly reducing manual effort. However, when target programs contain numerous memory operations, effectively evaluating the risk level of each target site remains challenging. By establishing an assessment metric to quantify the probability of memory leaks at PML/Risk Areas, the efficiency of our approach could be substantially improved. Furthermore, the assessment mtric will facilitate more informed seed selection. This enhancement is particularly valuable because our current scheduling scheme primarily addresses seed starvation but lacks comprehensive seed quality assessment.

6.2 Technical Generalizability

While the primary focus of our approach is memory leak detection, the proposed solution is theoretically applicable to most multi-target directed fuzzing scenarios. In future work, reducing system coupling is expected to enhance generalization capability, thereby extending the framework's applicability to broader vulnerability discovery tasks, particularly Use-After-Free (UAF) detection. However, it is crucial to develop vulnerability-specific guidance strategies - for instance, enforcing strict temporal ordering between use and free operations when detecting UAF vulnerabilities.

6.3 Ready for Deployment

Although we do not illustrate how RBZZER might be deployed in a real software development lifecycle, we consider that RBZZER is not only effective on open-source programs but also can be deployed in a real software development lifecycle. RBZZER is a standard grey-box fuzzer designed to enhance software security through fuzzing like any other grey-box fuzzer. It specializes in dynamic testing of software functional modules, interfaces, and referenced libraries to detect vulnerabilities, particularly memory leaks. As a CI/CD-compatible solution, RBZZER operates in two critical phases: (1) During continuous integration (CI), it automatically triggers fuzzing upon code commits or merges to the main branch, complementing static code analysis and unit testing; (2) In continuous deployment (CD), it performs fuzzing on generated instrumented binaries or APIs post-build.

7 Conclusion

This paper addresses the aimlessness issue in memory leak vulnerability fuzz testing by introducing directed fuzz testing technology, presenting a memory leak-directed guidance mechanism based on deviation probability; in response to the starvation problem of memory leak points, a seed scheduling strategy based on risk areas is proposed. We implemented and named the above methods as RBZZER. Experimental results show that RBZZER finds 52% more program unique crashes than the second-best counterpart. In particular, RBZZER can discover the amount of memory leakage at least 112% more than the other baseline fuzzers. Besides, RBZZER detects memory leaks at an average speed that is 9.10x faster than MemLock.

Acknowledgement: The authors are grateful for the highly constructive engagement by the reviewers and editors.

Funding Statement: This work is supported by the National Key R&D Program of China (No. 2021YFB3101803).

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Xi Peng, Peng Jia; data collection: Xi Peng; analysis and interpretation of results: Xi Peng, Ximing Fan, Jiayong Liu; draft manuscript preparation: Xi Peng, Jiayong Liu. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are available from the corresponding author, Peng Jia, upon reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Michal Z. American fuzzy lop. 2013. [cited 2025 May 21]. Available from: https://lcamtuf.coredump.cx/afl/.
2. Hodován R, Vince D, Kiss K. Fuzzing javascript environment APIs with interdependent function calls. In: Ahrendt W, Tapia Tarifa S, editors. Integrated formal methods. IFM 2019. Lecture notes in computer science. 2019. Vol. 11918. Cham, Switzerland: Springer. doi:10.1007/978-3-030-34968-4_12.
3. Zhou C, Zhang Q, Wang M, Guo L, Liang J, Liu Z, et al. Minerva: browser API fuzzing with dynamic mod-ref analysis. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering; Singapore; 2022. p. 1135–47. doi:10.1145/3540250.3549107.
4. Scharnowski T, Bars N, Schloegel M, Gustafson E, Muench M, Vigna G, et al. Fuzzware: using precise MMIO modeling for effective firmware fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22); 2022; Boston, MA, USA [cited 2025 May 21]. p. 1239–56. Available from: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski.
5. Li W, Shi J, Li F, Lin J, Wang W, Guan L. µAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware. In: Proceedings of the 44th International Conference on Software Engineering; Pittsburgh, PA, USA; 2022. p. 1–12. doi:10.1145/3510003.3510208.
6. Kim H, Jeong Y, Choi W, Lee DH, Jo HJ. Efficient ECU analysis technology through structure-aware CAN fuzzing. IEEE Access. 2022;10(260):23259–71. doi:10.1109/ACCESS.2022.3151358.
7. Böhme M, Pham VT, Nguyen MD, Roychoudhury A. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security; Dallas, TX, USA; 2017. p. 2329–44. doi:10.1145/3133956.3134020.
8. Godefroid P, Kiezun A, Levin MY. Grammar-based whitebox fuzzing. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation; Tucson, AZ, USA; 2008. p. 206–15. doi:10.1145/1375581.1375607.
9. Godefroid P, Levin MY, Molnar DA. Automated whitebox fuzz testing. NDSS. 2008 [cited 2025 May 21]; 8:151–66. Available from: https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/.
10. Liu Y, Li Y, Deng G, Liu Y, Wan R, Wu R, et al. Morest: model-based RESTful API testing with execution feedback. In: Proceedings of the 44th International Conference on Software Engineering; Pittsburgh, PA, USA; 2022. p. 1406–17. doi:10.1145/3510003.3510133.
11. Tsai CH, Tsai SC, Huang SK. REST API fuzzing by coverage level guided blackbox testing. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS); 2021; Hainan, China: IEEE. p. 291–300. doi:10.1109/QRS54544.2021.00040.
12. Mansur MN, Christakis M, Wüstholz V, Zhang F. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2020. p. 701–12. doi:10.1145/3368089.3409763.
13. Liang J, Wang M, Zhou C, Wu Z, Jiang Y, Liu J, et al. PATA: fuzzing with path aware taint analysis. In: 2022 IEEE Symposium on Security and Privacy (SP); San Francisco, CA, USA: IEEE; 2022. p. 1–17. doi:10.1109/SP46214.2022.9833594.
14. Wu M, Jiang L, Xiang J, Huang Y, Cui H, Zhang L, et al. One fuzzing strategy to rule them all. In: Proceedings of the 44th International Conference on Software Engineering; Pittsburgh, PA, USA; 2022. p. 1634–45. doi:10.1145/3510003.3510174.
15. Wang J, Chen B, Wei L, Liu Y. Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP); San Jose, CA, USA: IEEE; 2017. p. 579–94. doi:10.1109/SP.2017.23.
16. Wang Y, Jia X, Liu Y, Zeng K, Bao T, Wu D, et al. Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: Network and Distributed Systems Security (NDSS) Symposium; 2020 Feb; San Diego, CA, USA. p. 23–6. doi:10.14722/ndss.2020.24422.
17. Chen Y, Zhong R, Hu H, Zhang H, Yang Y, Wu D, et al. One engine to fuzz'em all: generic language processor testing with semantic validation. In: 2021 IEEE Symposium on Security and Privacy (SP); San Francisco, CA, USA: IEEE; 2021. p. 642–58. doi:10.1109/SP40001.2021.00071.

18. MITRE. CVE-2018-17985; 2018 [cited 2025 May 21]. Available from: https://www.cve.org/CVERecord?id=CVE-2018-17985.
19. MITRE. CVE-2019-6262; 2019 [cited 2025 May 21]. Available from: https://www.cve.org/CVERecord?id=CVE-2019-6262.
20. Islam U, Muhammad A, Mansoor R, Hossain MS, Ahmad I, Eldin ET, et al. Detection of distributed denial of service (DDoS) attacks in IOT based monitoring system of banking sector using machine learning models. Sustainability. 2022;14(14):8374. doi:10.3390/su14148374.
21. Wu W, Chen Y, Xu J, Xing X, Gong X, Zou W. FUZE: towards facilitating exploit generation for kernel Use-After-Free vulnerabilities. In: 27th USENIX Security Symposium (USENIX Security 18); 2018 [cited 2025 May 21]; Baltimore, MD, USA. p. 781–97. Available from: https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei.
22. Zheng Y, Li Y, Zhang C, Zhu H, Liu Y, Sun L. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis; 2022. p. 417–28. doi:10.1145/3533767.3534414.
23. Wang H, Xie X, Li Y, Wen C, Li Y, Liu Y, et al. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering; Seoul, Republic of Korea; 2020. p. 999–1010. doi:10.1145/3377811.3380386.
24. Yu Y, Jia X, Liu Y, Wang Y, Sang Q, Zhang C, et al. HTFuzz: heap operation sequence sensitive fuzzing. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering; Rochester, MI, USA; 2022. p. 1–13. doi:10.1145/3551349.3560415.
25. Lee G, Shim W, Lee B. Constraint-guided directed greybox fuzzing. In: 30th USENIX Security Symposium (USENIX Security 21); 2021 [cited 2025 May 21]. p. 3559–76. Available from: https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu.
26. Wen C, Wang H, Li Y, Qin S, Liu Y, Xu Z, et al. Memlock: memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering; Seoul, Republic of Korea; 2020. p. 765–77. doi:10.1145/3377811.3380396.
27. Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security; Vienna, Austria; 2016. p. 1032–43. doi:10.1145/2976749.2978428.
28. Lemieux C, Padhye R, Sen K, Song D. Perffuzz: automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis; Amsterdam, Netherlands; 2018. p. 254–65. doi:10.1145/3213846.3213874.
29. Yun I, Lee S, Xu M, Jang Y, Kim T. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18); 2018 [cited 2025 May 21]; Baltimore, MD, USA. p. 745–61. Available from: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.
30. MITRE. CWE:401: missing release of memory after effective lifetime. [cited 2025 May 21]. Available from: https://cwe.mitre.org/data/definitions/401.html.
31. Yu Z, Liu Z, Cong X, Li X, Yin L. Fuzzing: progress, challenges, and perspectives. Comput Mater Contin. 2024;78(1):1–29. doi:10.32604/cmc.2023.042361.
32. MITRE. CVE-2019-20023. 2019 [cited 2025 May 21]. Available from: https://www.cve.org/CVERecord?id=CVE-2019-20023.
33. GNOME. libxml2. 2024 [cited 2025 May 21]. Available from: https://gitlab.gnome.org/GNOME/libxml2.
34. JasPer. jasper. 2024 [cited 2025 May 21]. Available from: https://github.com/jasper-software/jasper/tree/release-2.0.
35. Patrice L, Sébastien LA. Xpdf-4.00; 2024 [cited 2025 May 21]. Available from: https://github.com/kermitt2/xpdf-4.00.
36. DynamoRio. DynamoRio; 2024 [cited 2025 May 21]. Available from: https://github.com/DynamoRIO/dynamorio.
37. Gaasedelen M. Lighthouse; 2024 [cited 2025 May 21]. Available from: https://github.com/gaasedelen/lighthouse.

38. Du Z, Li Y, Liu Y, Mao B. Windranger: a directed greybox fuzzer driven by deviation basic blocks. In: Proceedings of the 44th International Conference on Software Engineering; Pittsburgh, PA, USA; 2022. p. 2440–51. doi:10.1145/3510003.3510197.
39. Li Y, Ji S, Chen Y, Liang S, Lee WH, Chen Y, et al. UNIFUZZ: a holistic and pragmatic metrics-driven platform for evaluating fuzzers. In: 30th USENIX Security Symposium (USENIX Security 21); 2021 [cited 2025 May 21]. p. 2777–94. Available from: https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei.
40. Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, et al. Collafl: path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP); 2018; Francisco, CA, USA: IEEE. p. 679–96. doi:10.1109/SP.2018.00040.
41. Klees G, Ruef A, Cooper B, Wei S, Hicks M. Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security; Toronto, ON, Canada; 2018. p. 2123–38. doi:10.1145/3243734.3243804.
42. Lemieux C, Sen K. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering; Montpellier, France; 2018. p. 475–85. doi:10.1145/3238147.3238176.
43. Foundation FS. GNU Binutils; 2024 [cited 2025 May 21]. Available from: https://github.com/bminor/binutils-gdb/tree/users/hjl/linux/release/2.29.51.0.1.
44. libming. libming; 2024 [cited 2025 May 21]. Available from: https://github.com/libming/libming/tree/ming-0_4_8.
45. Systems A. Bento4; 2024 [cited 2025 May 21]. Available from: https://github.com/axiomatic-systems/Bento4/tree/v1.5.1-620.
46. Arcuri A, Briand L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering; Waikiki, HI, USA; 2011. p. 1–10. doi:10.1145/1985793.1985795.
47. Fan G, Wu R, Shi Q, Xiao X, Zhou J, Zhang C. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE); Montreal, QC, Canada: IEEE; 2019. p. 72–82. doi:10.1109/ICSE.2019.00025.
48. Li W, Cai H, Sui Y, Manz D. PCA: memory leak detection using partial call-path analysis. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2020. p. 1621–5. doi:10.1145/3368089.3417923.
49. Cao S, Sun X, Bo L, Wu R, Li B, Wu X, et al. Learning to detect memory-related vulnerabilities. ACM Transactions on Software Engineering and Methodology. 2023;33(2):1–35. doi:10.1145/3624744.
50. Cerion AB, Christian B, Jeremy F, Paul F, Tom H, Petar J et al. Valgrind; 2024 [cited 2025 May 21]. Available from: https://valgrind.org/.
51. Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, et al. Hawkeye: towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security; Toronto, ON, Canada; 2018. p. 2095–108. doi:10.1145/3243734.3243849.
52. Huang H, Guo Y, Shi Q, Yao P, Wu R, Zhang C. Beacon: directed grey-box fuzzing with provable path pruning. In: 2022 IEEE Symposium on Security and Privacy (SP); San Francisco, CA, USA: IEEE; 2022. p. 36–50. doi:10.1109/SP46214.2022.9833751.
53. Yu L, Lu Y, Shen Y, Li Y, Pan Z. Vulnerability-oriented directed fuzzing for binary programs. Sci Rep. 2022;12(1):4271. doi:10.1038/s41598-022-07355-5.
54. Koffi KA, Kampourakis V, Kolias C, Song J, Ivans RC. Speeding-up fuzzing through directional seeds. Int J Inf Secur. 2025;24(2):77. doi:10.1007/s10207-024-00953-6.