Computers & Security 159 (2025) 104691

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

DynamicFuzz: Confidence-based directed greybox fuzzing for programs with unreliable call graphs

Hao Jiang, Kang Wang, Yujie Yang, Shan Zhong, Shuai Zhang, Chengjie Liu, Xiarun Chen, Weiping Wen *

Peking University, Beijing, China

ARTICLE INFO

Keywords:
Directed greybox fuzzing
Indirect call relations
Adaptable call graph

ABSTRACT

Directed greybox fuzzing (DGF) is a security testing technique designed to test specific targets. Current DGF techniques face challenges due to the dynamic nature of indirect calls. The main challenges include mitigating the influence of indirect call omissions and misjudgments on seed guidance and guiding fuzzing on unreliable function call graphs.

This paper introduces DynamicFuzz, a novel dynamic guidance mechanism that uses the confidence of indirect calls to update the call graph and adjust path priorities during fuzzing. Our key insight is that functions connected by indirect calls tend to form function islands in the call graph. These islands help focus fuzzing on critical areas, improving both guidance efficiency and control over complex program structures. DynamicFuzz also incorporates two depth metrics – function depth and island depth – to better estimate the importance of each path. Based on this, DynamicFuzz employs four guiding strategies: the Target Function Selection Strategy, the Function Island Prioritization Strategy, the High-Confidence Path Prioritization Strategy, and the Deep Indirect Call Prioritization Strategy. These strategies allow DynamicFuzz to guide fuzzing effectively even when the call graph is unreliable. We evaluate DynamicFuzz on 17 benchmarks from three test suites. Compared to AFLGo, AFL, and FairFuzz, it reaches target locations 5.64× , 3.01× , and 2.89× faster, and detects target crashes 69.8× , 48.37× , and 161.20× faster, respectively. Additionally, DynamicFuzz discovered 8 CVEs from the real world.

1. Introduction

Fuzzing has become one of the most effective and widely adopted techniques for automated software vulnerability detection (Blair et al., 2020; Böhme et al., 2016; Chen et al., 2020a). Fuzzing tools generate new seed inputs by leveraging runtime feedback and initial seed inputs from the Program Under Test (PUT), potentially exposing erroneous behaviors. Currently, most fuzzers focus on overall code coverage to uncover potential vulnerabilities (Chen and Chen, 2018; Böhme et al., 2016). However, traditional fuzzing tests often fail to reach predefined target areas. In 2017, the Directed Greybox Fuzzing (DGF) AFLGo (Böhme et al., 2017) was proposed. AFLGo allocates a significant portion of its time budget to reaching and testing target sites. Consequently, in specific scenarios – such as patch testing, vulnerability reproduction, and verification of potential erroneous code – DGF demonstrates advantages over Coverage Guided Fuzzing (CGF) (Wang et al., 2020b).

To achieve directedness, DGF collects and processes relevant information during seed execution, including seed distance (Böhme et al., 2017), covered function similarity (Chen et al., 2018), path probability (Lin et al., 2023), data-flow graphs (Du et al., 2022), and sequence coverage (Liang et al., 2019). Despite these achievements, existing DGF techniques still face two challenges:

Challenge 1: Mitigating the Influence of Indirect Call Omissions and Misjudgments on Seed Guidance. Approaches such as Hawkeye (Chen et al., 2018), Savior (Chen et al., 2020b), and RLTG (He and Zhu, 2023) employ inclusion-based pointer analysis (Andersen, 1994), while AFLGopher (Bai et al., 2023) uses a multi-layer type analysis approach (Lu and Hu, 2019) to identify Indirect Calls (ICs). However, due to the dynamic characteristics of function pointers, these methods often lead to significant omissions and misjudgments. Such inaccuracies can result in misleading seed guidance. For example, as shown in Fig. 1, $Function\{2,3,5,8\}$ involves ICs. Consider three seeds, A, B,

* Corresponding author.
E-mail addresses: soundfuture@stu.pku.edu.cn (H. Jiang), kangwang@stu.pku.edu.cn (K. Wang), 2501210755@stu.pku.edu.cn (Y. Yang), fourilace@stu.pku.edu.cn (S. Zhong), shuai_zh@stu.pku.edu.cn (S. Zhang), cheng@stu.pku.edu.cn (C. Liu), xiar_c@pku.edu.cn (X. Chen), weipingwen@pku.edu.cn (W. Wen).

https://doi.org/10.1016/j.cose.2025.104691
Received 17 April 2025; Received in revised form 8 July 2025; Accepted 29 September 2025
Available online 1 October 2025

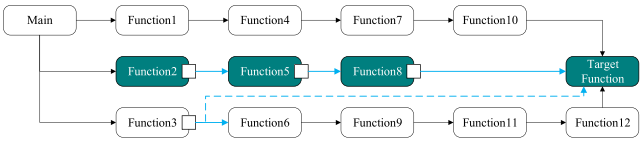

Fig. 1. Impact of indirect call identification errors on seed guidance. Squares represent indirect call sites. Black lines indicate direct calls, blue lines indicate indirect calls, and dashed blue lines represent incorrectly identified indirect calls. The optimal path is highlighted in dark green. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and C, whose seed paths are $\langle Main, Function1\rangle$, $\langle Main, Function2\rangle$, and $\langle Main, Function3\rangle$, respectively. If any Indirect Call Edge (ICE) on $Function\{2, 5, 8\}$ is missed, the optimal seed B would be disregarded. Conversely, if $Function3$ falsely reports an ICE reaching the Target Function (TF), the least suitable seed C would be advanced. Therefore, mitigating the influence of indirect call omissions and misjudgments on seed guidance remains a challenge.

Challenge 2: Guiding Fuzzing Test on Unreliable Function Call Graphs. Currently, DGF relies on the Call Graph (CG) for distance calculation. However, existing studies (Bai et al., 2023; Böhme et al., 2017; Chen et al., 2020b; He and Zhu, 2023; Cheng et al., 2024) have not been able to construct a completely reliable CG. If omissions occur in ICs, the CG might lack paths leading to the TF, resulting in guidance deviation or even failure. Conversely, if misjudgments occur, DGF may be misguided by false paths, resulting in resource wastage. Consequently, it is a challenge to improve current algorithms to enable DGF to effectively guide fuzzing, even when the CG is unreliable.

This paper introduces DynamicFuzz, a novel solution designed to address two challenges. For Challenge 1, we introduce an indirect call elimination mechanism that gradually enhances the authenticity of the CG and reduces the impact of errors in indirect call analysis on seed guidance. DynamicFuzz first generates a basic CG through static analysis and then analyzes possible ICEs using the state-of-the-art indirect call analysis tool SEA (Cheng et al., 2024). Each ICE identified by SEA is assigned an initial confidence score. During fuzzing, the confidence of unexecuted ICEs gradually decreases. When calculating seed distance, edges with lower confidence are assigned longer distances, thus having less influence on seed guidance. While fuzzing, DynamicFuzz also records seed paths and fixes the confidence of the traversed ICEs to 1, thereby preserving authentic ICEs and eliminating spurious ones. For clarity, we define paths that consist solely of direct calls and ICEs with a confidence score of 1 as stable paths, while categorizing all other paths as unstable paths.

For Challenge 2, we introduce the concept of function islands and propose four guiding strategies based on function islands, island depth, function depth, and the confidence of ICE to achieve efficient fuzzing guidance. A function island refers to a group of functions organized according to ICE, while the number of ICEs present in reachable paths determines both function depth and island depth. DynamicFuzz first identifies all paths from the CG leading to the TF. Next, DynamicFuzz computes the function depth of the TF and the island depth of the function island to which the TF belongs. Finally, DynamicFuzz applies the following four guiding strategies:

(1) Target Function Selection Strategy: This strategy identifies all functions that can be reached through stable paths and contain ICs as guidance targets. The aim is to efficiently verify the validity of ICEs and enhance the accuracy of the CG. Consequently, all functions containing ICs are regarded as Relative Target Functions ($RTFs$).

(2) Function Island Prioritization Strategy: This strategy prioritizes the exploration of function islands that are close in depth to the island containing the TF and that include more RTF. This strategy is based on two observations: (i) When two function islands have similar depths, the likelihood of them being connected through indirect calls increases. (ii) When one RTF serves as a target for fuzzing, other $RTFs$ within the same function island can often be fuzzed simultaneously.

(3) High-Confidence Path Prioritization Strategy: This strategy prioritizes exploring TF-Reachable Paths that include high-confidence ICEs. Paths with high-confidence ICEs are more likely to connect to the TF than uncertain paths.

(4) Deep Indirect Call Prioritization Strategy: When no reachable path to the TF is found in the CG, this strategy prioritizes exploring deeper ICEs. This strategy is based on the following observations: the caller function usually performs parameter checking and preprocessing. In contrast, the callee function is more focused on data processing, making it more prone to exposing vulnerabilities. Additionally, deeper ICEs are often less explored, which presents a higher potential for reaching the TF.

The main contributions of this paper are summarized as follows:

- For Challenge 1, we propose a method for dynamically updating the call graph that introduces an indirect call elimination mechanism. This approach balances accuracy and efficiency in updating indirect call edges while minimizing the impact of analysis errors on seed guidance.
- For Challenge 2, we propose four guiding strategies based on function island, depth information, and indirect call confidence. Together, these strategies provide effective guidance in unreliable call graphs and improve the efficiency of reaching the target function.
- We propose a seed sorting algorithm based on execution count and seed weight, which effectively enhances the efficiency of reaching target functions.
- We implemented a tool that organically integrates these ideas and evaluates them in crash replication and target arrival. Additionally, DynamicFuzz discovered eight undisclosed vulnerabilities in real-world programs.

2. Related work

This section highlights two significant methodologies: directed greybox fuzzing and coverage-guided greybox fuzzing.

2.1. Directed greybox fuzzing

AFLGo (Böhme et al., 2017) is the first directed greybox fuzzing tool. It prioritizes seeds that are closer to predefined targets by calculating the distance between each seed and the targets, thereby minimizing this distance. Based on the idea of AFLGo (Böhme et al., 2017), Hawkeye (Chen et al., 2018) proposed tracking similarity and adapted seed prioritization, power scheduling, and mutation strategies to enhance guidance. Similarly, most subsequent research in DGF has focused on improving the speed of guidance while paying less attention to achieving accurate guidance under incomplete or unreliable CG.

Many DGFs including AFLGo (Böhme et al., 2017), Beacon (Huang et al., 2022), WindRanger (Du et al., 2022), HyperGo (Lin et al., 2023), AFLRun (Rong et al., 2024), WAFLGo (Xiang et al., 2024), and OptFuzz (Wang et al., 2024) have not specifically addressed the challenge of indirect calls. In contrast, tools such as Hawkeye (Chen et al., 2018), SAVIOR (Chen et al., 2020b), RLTG (He and Zhu, 2023), AFLGopher (Bai et al., 2023), SelectFuzz (Luo et al.), SDFuzz (Li et al., 2024), LibAFL-DiFuzz (Parygina et al., 2024) attempt to fill this gap using static methods like pointer analysis or multi-level type analysis. However, due to the inherent limitations of static analysis, these tools often suffer from high false-positive and false-negative rates when dealing with large-scale programs that contain ICs, thereby affecting the effectiveness of fuzzing guidance.

The root cause of such inaccuracies lies in the difficulty of predicting the dynamic behavior of function pointers through static analysis. Tools

Table 1
Distribution of indirect calls in real programs.

Project	Version	Functions	Calls	Indirect calls	Proportion
FreeImage	3.18.0	30,009	142,480	3,088	2.17%
libming	0.4.8	4,807	24,464	547	2.24%
libxml2	2.9.2	9,679	67,926	2,989	4.40%
binutils	2.26	11,397	60,289	3,085	5.12%
giflib	5.1	609	2,521	19	0.80%
mjs	1.26	435	4,325	126	2.98%
freetype	2.13.2	5,555	70,866	1,462	2.10%
libjpeg-turbo	3.0.3	8,175	125,891	5,726	4.55%
Average	–	8,833	62,345	2,130	3.12%

such as ParmeSan (Österlund et al., 2020), AFLTeam (Pham et al., 2021), and DAFL (Kim et al., 2023) attempt to address this issue by collecting function call information at runtime and dynamically updating the CG. However, these approaches neither fully leverage the results of existing indirect call recognition tools nor validate the accuracy of the constructed CGs. As a result, they may fail to identify the optimal path to the TF or may be misled by incorrect ICEs during fuzzing guidance. DynamicFuzz addresses this challenge by dynamically constructing the CG and adaptively adjusting both the fuzzing targets and the confidence of ICEs, thereby providing a flexible and effective solution.

2.2. Coverage-guided greybox fuzzing

As a fuzzing technique, CGF aims to guide testing to achieve broader code coverage. For instance, AFL is a classic example of CGF that leverages coverage feedback to direct seed mutation. To increase coverage, hybrid fuzzing (Godefroid et al., 2005, 2008; Cha et al., 2015) incorporates symbolic execution to solve conditional constraints in PUTS rapidly. In addition to constraint-solving techniques, other methods focus on improving seed scheduling (Böhme et al., 2016; Li et al., 2019; Wang et al., 2021) and detecting specific types of vulnerabilities (Wang et al., 2020a; Wen et al., 2020). Although the goal of DynamicFuzz is to reach the target as efficiently as possible rather than increasing coverage, DynamicFuzz can still benefit from some of the CGF techniques.

3. Motivation

Indirect calls are widely used in real-world programs, particularly in C/C++ programs. ICs often occur when function pointers are invoked. We analyzed the prevalence of ICs in eight popular programs, detailed in Table 1.

Table 1 presents the project's basic information and statistical results. It includes the number of functions (*Functions*), function calls (*Calls*), indirect calls (*Indirect Calls*), and the proportion of indirect calls in function calls (*Proportion*). Although the proportion of indirect calls is relatively low, our sampling investigation reveals that applications often utilize indirect calls to implement modular architectures and callback mechanisms. As a critical link in function dispatch, indirect calls connect different modules to form key paths. Therefore, neglecting indirect calls may lead to significant deficiencies in the CG.

Fig. 2 shows the CG associated with CVE-2023-47992, highlighting codes related to indirect calls and the vulnerability. The CG is divided into three levels of depths: 0, 1, and 2, according to indirect calls. The CVE-2023-47992 triggers when $LoadPixelData$ invokes $\_MemoryReadProc$ with a large unsigned integer parameter, leading to an integer overflow during type conversion. The function $ReadMemory$ checks the $config\_stream$ variable before calling $FreeImage\_ReadMemory$. If $config\_stream$ is $FALSE$, parameter $stream$ becomes $NULL$, preventing $FreeImage\_ReadMemory$ from calling $\_MemoryReadProc$.

3.1. Limitations of the existing approach

There are three situations to consider when reproducing this CVE with DGF: (1) If the indirect call edges are not pre-constructed, a valid path from $Main$ to $\_MemoryReadProc$ cannot be found, resulting in the inability to compute the distance. (2) If the indirect call edges are pre-built but contain omissions or misreporting, as discussed in Section 1, this can similarly lead to errors in distance computation, causing bias in the guidance. (3) Even when indirect call edges are correctly constructed, exploiting a vulnerability typically depends on a specific execution path. Consequently, indirect call edges that are only reachable under certain configurations – or those that are shorter yet unrelated to the vulnerability – may disrupt the analysis. For example, the path <$Main$, $ReadMemory$, $FreeImage\_ReadMemory$, $\_MemoryReadProc$> is shorter than the dark green path. Suppose seeds A and B correspond to <$Main$, $FreeImage\_LoadFromHandle$> and <$Main$, $ReadMemory$>, respectively. In that case, the analysis will prioritize seed B's shorter path, even if it is unreachable under default configurations ($config\_stream$ is $FALSE$) and irrelevant to the vulnerability, thereby reducing fuzzing efficiency.

3.2. Our observations and methods

In this case, ensuring the authenticity of ICEs is critical for guiding fuzzing. Research (Zhu et al., 2021) indicates that the DGF-based graph constructor can uncover an additional 62.9% of reachable functions compared to the static analyzer. Based on this finding, we optimize our fuzzing strategies by collecting execution path data and dynamically adjusting the guidance. The sections below describe how each of our four guiding strategies works.

Initially, when constructing the CG, all identified ICEs are assigned high confidence, although their authenticity remains to be verified. Based on Guiding Strategy 1, every function containing ICs is set as a target to distinguish true ICEs from false ones rapidly. Under this strategy, since $FreeImage\_LoadFromHandle$ and $ReadMemory$ are the $RTFs$, DynamicFuzz will focus on increasing their coverage rather than wasting resources on non-RTF functions such as $FreeImage\_Open$, $FreeImage\_GetFIFCount$, and $FindNodeFromFif$.

As fuzzing progresses, the ICEs between $FreeImage\_LoadFromHandle$ and several $Load$ functions are gradually verified, along with the ICE between $ReadMemory$ and $FreeImage\_ReadMemory$. Verifying ICEs not only connects previously isolated regions in the CG but also lowers the confidence of falsely reported ICEs, effectively pruning false paths. During this process, we observed that some functions connect solely through ICEs. For example, verifying the ICE between $FreeImage\_LoadFromHandle$ and $Load_3$ transforms some functions that were unreachable by stable paths into functions that are reachable by stable paths. These sets of functions, linked by ICEs, create what we refer to as "function islands". This concept is crucial for our guidance strategy and will be explicitly defined in Section 4.2. Each island is connected by ICEs, while the functions within each island are linked through direct calls. Fig. 3 illustrates the function island view associated with CVE-2023-47992.

Based on guiding strategy 2, the function islands containing $LoadPixelData$ and $FreeImage\_ReadMemory$ are categorized with a depth of 1, while those that include $FreeImage\_LoadFromHandle$ and $ReadMemory$ are assigned a depth of 0. Consequently, islands at depth 1 are explored first. This strategy directs the fuzzer's attention to regions nearer to the TF, accelerating the discovery process.

Based on guiding strategy 3, $Load$ functions, except for $Load_3$, do not provide paths to the TF and, therefore, receive fewer fuzzing resources. Since $FreeImage\_ReadMemory$ cannot reach $\_MemoryReadProc$ under default configurations, its ICE confidence decreases as fuzzing proceeds. In contrast, $LoadPixelData$ successfully connects to $\_MemoryReadProc$, establishing a stable path that is prioritized to fuzz until the vulnerability is uncovered.

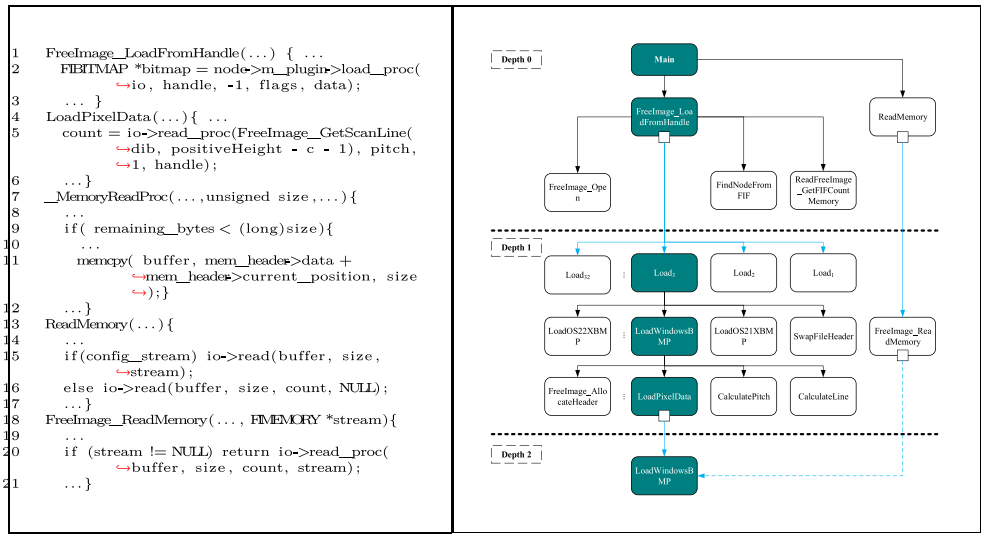

Fig. 2. The CG and codes related to CVE-2023-47992. The vulnerability-triggering path is highlighted in green. Lines and squares have the same meaning as Fig. 1. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

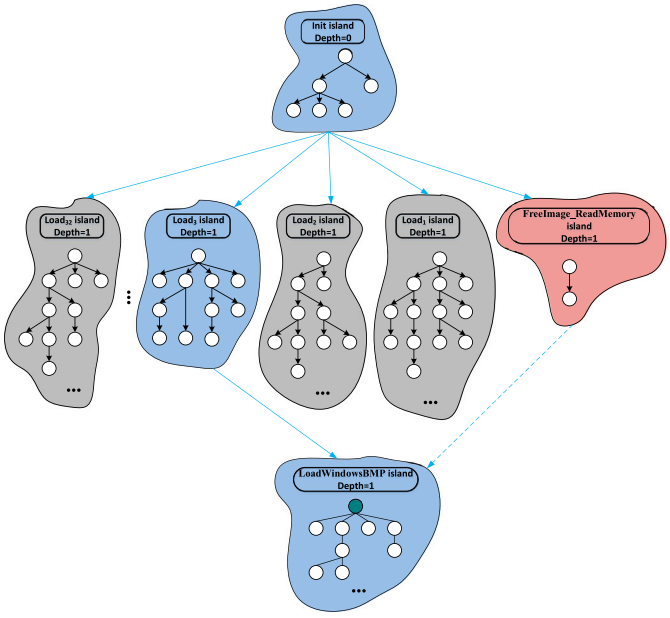

Fig. 3. The Island related to CVE-2023-47992. Function islands with a blue background represent those traversed by the vulnerability-triggering path, while islands with gray and red backgrounds have been pruned by guiding strategy 3. Deep green function nodes represent the target functions. Lines have the same meaning as Fig. 1. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Finally, according to Guiding Strategy 4, when the indirect call analysis tool fails to detect particular ICEs, resulting in no paths in the CG that are reachable to the TF, DynamicFuzz addresses this guidance failure by exploring the ICs using a Depth-First Search approach. Specifically, DynamicFuzz prioritizes exploring the $LoadPixelData$ and $FreeImage\_ReadMemory$ after establishing stable paths to these two deeper functions. The function $FreeImage\_LoadFromHandle$ acts as a dispatch function, indirectly invoking the appropriate handlers through function pointers, while $LoadPixelData$ serves as a data loader responsible for validating and processing the image header structure. It also calls $\_MemoryReadProc$ to read the relevant data. As the call depth increases, memory read and write operations become more frequent, which raises the likelihood of vulnerabilities. Therefore, the effectiveness of this strategy is evident.

4. Approach of DynamicFuzz

In this section, we provide an overview of the components of DynamicFuzz and define the concept of function islands. Additionally, we discuss depth computation, weight computation, seed sorting algorithm, and scheduling methods. Fig. 4 illustrates the architecture of DynamicFuzz, which mainly consists of two parts: the Static Analysis module and the Fuzzing module. The static analysis is used to instrument the program, extract function and IC information, and construct the CG. Moreover, similar to CGF, the tool also collects edge coverage data to guide the exploration process.

During the fuzzing process, DynamicFuzz executes seeds, extracts their execution queues, and constructs seed paths through path analysis. It then evaluates these seed paths to determine whether they should be retained. If a path contains an edge that does not exist in the CG, the edge is recognized as an ICE. Then, the new edge, along with its associated function island, is updated in the CG. Additionally, DynamicFuzz updates the depth and weight information for both function nodes and function islands. It then computes the weight and priority of each seed and inserts them into a priority queue. Once the optimal seed is selected from the queue and power scheduling is allocated, DynamicFuzz mutates the seed and executes the newly generated seed.

In the following sections, we will answer the following questions in detail:

(1) How does DynamicFuzz instrument programs (Section 4.1);
(2) What is the definition of a function island (Section 4.2);
(3) How does DynamicFuzz calculate depth information (Section 4.3);
(4) How does DynamicFuzz calculate weight information (Section 4.4);
(5) How does DynamicFuzz perform seed sorting (Section 4.5);
(6) How does DynamicFuzz perform power scheduling (Section 4.6);
(7) What is the optimization strategies of DynamicFuzz (Section 4.7)

4.1. Program instrumentation

DynamicFuzz captures critical coverage and execution path information by inserting additional code into the program. In this work, we propose an instrumentation method for collecting function execution

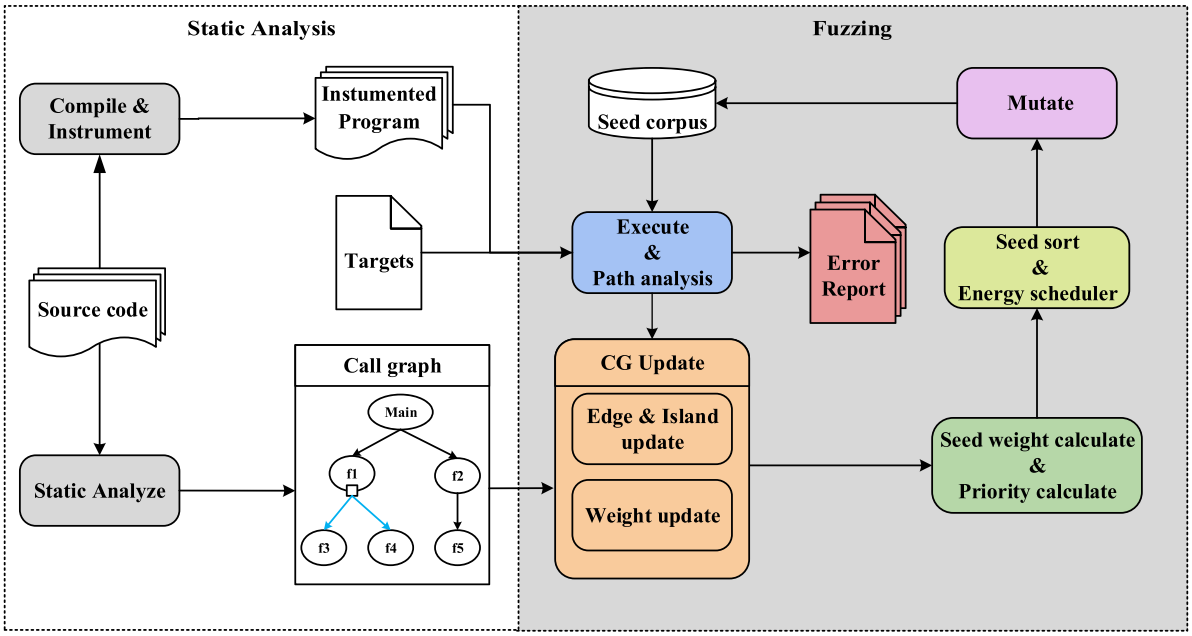

Fig. 4. DynamicFuzz architecture.

paths to improve the accuracy and efficiency of capturing seed execution paths. Specifically, DynamicFuzz inserts coverage collection code before the first instruction of each basic block and inserts execution path collection code before each function entry and each *ret* instruction. The coverage collection code is the same as that used in AFL, and it records the basic blocks and edge coverage during program execution to assist the DGF in discovering new execution paths. The execution path collection code records the call sequence of functions and ret instructions in a global queue during runtime, thereby facilitating the reconstruction of complete seed paths.

4.2. Definition of function island

A function island is a collection of functions clustered by their indirect call relationships. In Section 3, we present an illustrative example to provide an intuitive introduction to this concept. The following presents a more precise definition.

For two functions a and b in the CG, if there exists a reachable path from a to b, it is denoted as $R(a, b)$. If this path only contains direct call edges, then function a is said to be directly connected to function b, denoted as $R_D(a, b)$. If there exists a function node n and the ICE indegree of n is greater than 0, then the union of n and all function nodes directly connected by n is defined as a function island. The definition of a function island is given in Eq. (1).

$$I(n) = \{m \mid \forall m \in CG, R_D(n, m) = True\} \cup \{n\} \tag{1}$$

It is important to note that our definition allows function islands to overlap. As illustrated in Fig. 5, Island 2 encompasses both Island 3 and Island 4. This overlapping characteristic arises from the multi-affiliation nature of functions in real-world programs, where the same function can be triggered by ICs originating from different root nodes. This property is realized through a dynamic affiliation mechanism: function islands are created by verified ICEs rather than being statically bound to a single island. This approach more accurately reflects the semantics of real execution paths. DynamicFuzz leverages this diversity as part of its runtime adaptive guidance, permitting transient fluctuations in node weights as new execution information becomes available. This behavior is consistent with our core design philosophy: to continuously adjust the fuzzing direction in response to newly observed program behavior.

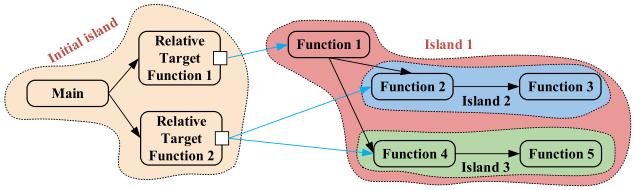

Fig. 5. Overlapping islands. Squares indicate indirect calls, black lines are direct call edges, and blue lines are indirect call edges. The cluster of functions in each area represents a function island. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

When constructing the CG, our approach strictly maintains the CG as a Directed Acyclic Graph (DAG) at all times. During the initial static construction, we employ a Depth-First Search to detect cycles. If a cycle is found, the checking edge is removed to break the cycle, thereby ensuring the CG's acyclic property. At runtime, whenever a new ICE missed by static analysis is discovered through seed execution, we provisionally add it to the graph and perform a cycle check (e.g., via topological sorting). If a cycle is formed, the new edge is discarded. While severing edges introduces a minor analysis inaccuracy, it does not affect the final seed ranking, as our evaluation of a seed is based on the function nodes it traverses.

4.3. Function depth and island depth

The depth of a function island and a function node indicates the level of ICs. After initializing the depths of the initial islands and function nodes, the depths of the remaining islands and functions can be calculated recursively. As described in Section 4.2, since the CG is maintained as a DAG, this recursive algorithm is guaranteed to terminate and converge in finite time.

The island that contains the input function is referred to as the initial island, denoted as I_{ini}. The depth of I_{ini} is set to 0, and each function node n in I_{ini} is also assigned a depth of 0. Since a function node may belong to multiple islands, its depth is calculated using a weighted average algorithm. The weighting algorithm proposed in this

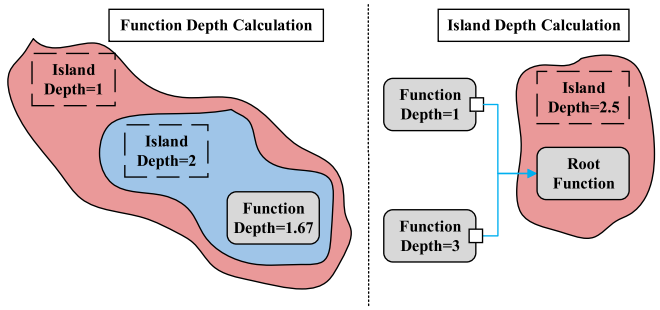

Fig. 6. Depth calculation schematic.

paper considers depths at higher levels to be more informative. The depth of a function node is calculated as shown in Eq. (2).

$$depN(n) = \begin{cases} 0 & \text{if } n \in I_{ini} \\ \sum_{I \in getI(n)} W_{id}(n, I) \times depI(I) & \text{else} \end{cases} \tag{2}$$

Here, $getI(n)$ denotes the set of islands to which the function node n belongs. $depI(I)$ denotes the depth of island I. $W_{id}(n, I)$ represents the relative depth of island I among all islands containing n, as given in Eq. (3).

$$W_{id}(n, I) = \frac{depI(I)}{\sum_{I' \in getI(n)} depI(I')} \tag{3}$$

Let F_r represent the root node of an island, and F_c represent the set of functions that indirectly invoke F_r. The depth of an island is determined by F_c and is computed as the weighted average of the depths of functions in F_c. Since at least one IC is involved, the depth should be incremented by 1. Hence, the formula for calculating the island's depth is given in Eq. (4):

$$depI(I(F_r)) = \begin{cases} 0 & \text{if } F_r \in I_{ini} \\ 1 + \sum_{f \in F_c} W_{nd}(f, F_c) \times depN(f) & \text{else} \end{cases} \tag{4}$$

The symbol $W_{nd}(F_c, f)$ denotes the relative depth of function f in F_c, as defined in Eq. (5)

$$W_{nd}(f, F_c) = \begin{cases} 0 & \text{if } \sum_{f' \in F_c} depN(f') = 0 \\ \frac{depN(f)}{\sum_{f' \in F_c} depN(f')} & \text{else} \end{cases} \tag{5}$$

The depth calculation is based on a weighted average of parent nodes or parent islands, rather than a single depth value. This design ensures that the influence of multi-path calls is taken into account. For instance, if an island triggered by a deep node is also accessed by a shallow caller, its depth will be dynamically balanced. This reflects the diversity of real program paths, where the same function can assume different roles in different contexts. Therefore, the depth calculation must synthesize the context from all call chains to avoid biases introduced by a single path. Fig. 6 illustrates scenarios of calculating island and function depths using this formula: for example, when a node belongs to two islands with depths of 1 and 2 respectively, its resulting depth is 1.67; similarly, when two functions with depths of 1 and 3 form an island through an indirect call, the island's depth becomes 2.5. In Section 5.5, we compare the depth calculation method with common algorithms to assess the effects of these approaches on fuzzing guidance.

4.4. Weighting algorithm

The core of our guidance system is a dynamic weighting algorithm that implements our four guiding strategies (mentioned in Section 3). This algorithm assigns weights to functions, islands, and seeds to guide the fuzzer. Algorithm 1 provides an overview of the weight update steps in our fuzzing process. DynamicFuzz first executes a seed and extracts its execution path. It then checks whether any new ICEs are found or verified. If so, the CG is updated, a new island is generated (line 3), and the island weights (line 4) as well as the target weights (line 5) are updated. DynamicFuzz then reverses the CG to back-propagate the new target weights to the connected function nodes, thereby updating the weights of the function nodes in the CG (line 7). If no new ICEs are found or verified, only the ICE confidence and the weights of the affected nodes are updated (lines 9–10). Finally, the seed weight is calculated based on the updated information (line 12).

In the following, we will describe how indirect call confidence, island weights, target weights, node weights, and seed weights are calculated.

Algorithm 1 Weight Calculating Algorithm

Input: $nodeWeights$, $TFWeights$, $islandWeights$, CG, $seed$
Output: $seedWeight$
1: $seedPath \leftarrow$ Exec($CG, seed$)
2: **if** hasNewEdge($seedPath$) **or** verifyNewICE($seedPath$) **then**
3: $newIsland, newTF \leftarrow$ updateMap($CG, seedPath$)
4: updateIslandWeights($CG, islandWeights, newIsland$)
5: updateTFWeights($islandWeights, TFWeights, newTF, CG$)
6: $reverseCG \leftarrow$ reverse(CG)
7: updateNodeWeights($reverseCG, nodeWeights, TFWeights$)
8: **else**
9: updateIndirectConfidence($CG, seedPath$)
10: updateNodeWeights($CG, nodeWeights, TFWeights$)
11: **end if**
12: $seedWeight \leftarrow$ calcSeedWeight($seedPath, nodeWeights, CG$)
13: **return** $seedWeight$

4.4.1. Indirect call confidence algorithm

The confidence of an ICE reflects our belief in its validity. The formula for computing the indirect call confidence is given in Eq. (6). Here, ice denotes an indirect call edge, and $IC(ice)$ denotes the call site of the ICE. $Func$ denotes the function to which the IC belongs, $ECount$ is the execution count of the function, and $CCount$ is the call count of the IC. The parameter α is used to control the decay rate. The calculation operates under a two-phase verification model to distinguish between scepticism about an entire IC and scepticism about a specific ICE.

Phase 1: IC Verification. If no potential ICE from a specific IC has been verified, the entire IC is considered "Speculative". The confidence of all ICEs originating from this IC decays uniformly based on the execution count of the function containing the IC. This phase aims to quickly reduce the credibility of ICs that might be false positives from static analysis and have never been reached during execution.

Phase 2: ICE Verification. Once any ICE from that IC is verified, the IC is considered valid. The system then transitions to this phase. The confidence of the remaining unverified ICEs from that site is now updated independently, based on the call count of the IC. The goal of this phase is to exclude invalid ICEs from a valid IC.

$$C_I(ice) = \begin{cases} 1 & \text{if } \exists p \in seedPaths, ice \in p \\ \alpha^{-(ECount(Func(IC(ice))))} & \text{else if } \forall i \in IC(ice), C_I(i) \neq 1 \\ \alpha^{-(CCount(IC(ice)))} & \text{else} \end{cases} \tag{6}$$

4.4.2. Islands weighting algorithm

To implement guiding strategy 2, we assign weights to the depths of the island. In this work, we use a normal distribution to calculate island depth weights because it naturally exhibits symmetry and a decaying characteristic: the weight peaks near the target function depth

and decreases as the depth deviates from this optimum. This approach avoids excessive resource allocation to islands with extreme depths and effectively focuses resources on the optimal depth range that is most likely to reach TF. The formula for computing the island depth weight is given in Eq. (7), where e denotes the natural constant, n represents the island's depth, and μ denotes the optimal depth. We determine the optimal depth μ using the depth value of the TF obtained during the initialization of the CG.

$$W_I^D(n) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(n-\mu)^2}{2}} \tag{7}$$

The island weight formula is defined in Eq. (8):

$$W_I(I) = \begin{cases} 1 - c^{|C_{ID}(I)| \times W_I^D(depI(I))} & I \notin getI(TF) \\ 10 & I \in getI(TF) \end{cases} \tag{8}$$

$C_{ID}(I)$ denotes the set of all ICs within island I. c is a constant ranging from 0 to 1. The default value of c is 0.2. For islands other than the target island, the weight is positively correlated with both the number of ICs within the island and its depth weight. We employ an exponential function to constrain the value between 0 and 1 and set the weight of the target island to 10 to emphasize its importance.

4.4.3. Target weighting algorithm

To implement guiding strategy 1, we consider functions containing ICs as guiding targets. Such functions are called Relative Target Functions (mentioned in Section 3). Together, the RTF and TF form the target set for directing the fuzzing process. To implement guiding strategy 4, we assign higher weights to function nodes with greater depth. Consequently, each RTF has two key attributes: the depth attribute and the island attribute. We combine these two attributes to calculate the target weight for each RTF. In practice, we set the weight of the TF to be 10 times the current maximum RTF weight. The weight formula for the target is shown in Eq. (9), where $MaxWRT$ denotes the highest weight among all $RTFs$.

$$W_T(n) = \begin{cases} (depN(n)+1) \times \sum_{I \in getI(n)} W_I(I) & \forall n \in CG, n \in RTFs \\ 10 \times MaxWRT & n = TF \end{cases} \tag{9}$$

4.4.4. Node weighting algorithm

Nodes refer to functions that are not targets. The reachable targets influence their weight. We propose a multi-factor node weight calculation method that combines indirect call confidence, function distance, and target weight to implement Guiding Strategy 3. To prioritize the exploration of ICs with high confidence, we convert the edge confidence into an edge weight in the CG. A lower edge confidence results in a higher edge weight. The conversion formula is given in Eq. (10)

$$W_E(edge) = \begin{cases} \frac{1}{C_I(edge)} & edge \in ICEs \\ 1 & \text{else} \end{cases} \tag{10}$$

Since edge confidence changes frequently, we use the Demetrescu–Italiano shortest path algorithm (Demetrescu and Italiano, 2005) to determine and update the distance between any two nodes in the CG, thereby reducing performance overhead. The node weight calculation formula is given in Eq. (11), where m denotes any node in the CG, T_f denotes the set of $RTFs$ and TFs, and D_{DI} represents the sum of the edge weights along the shortest path from m to p. As fuzzing progresses, the confidence in false ICEs gradually decreases, causing the edge weights to increase. Consequently, the node weights corresponding to paths formed by false ICEs continuously decrease, effectively eliminating errors in guidance caused by irrelevant nodes.

$$W_N(m) = \sum_{p \in R(m,T_f)} (W_T(p) \times (D_{DI}(m,p))^{-1}) \tag{11}$$

4.4.5. Seed weighting algorithm

DynamicFuzz quantifies the importance of seeds using seed weights. The fundamental property of a seed is a unique execution path. Therefore, the seed weight should depend on the function nodes it traverses. For any seed s, $P_N(s)$ denotes the nodes in the function-level path taken by seed s, while $P_T(s)$ denotes the $targets$ in the same path. Thus, the seed weight is as expressed in Eq. (14), where $\widetilde{W}_N(f)$ and $\widetilde{W}_T(f)$ represent min–max normalization for node weights and target weights, respectively. Additionally, $MinFW$ and $MaxFW$ indicate the minimum weight and maximum weight among all the functions.

$$\widetilde{W}_N(f) = \frac{W_N(f) - MinFW}{MaxFW - MinFW} \tag{12}$$

$$\widetilde{W}_T(f) = \frac{W_T(f) - MinFW}{MaxFW - MinFW} \tag{13}$$

$$W_S(s) = \sum_{f \in P_N(s)} \widetilde{W}_N(f) + \sum_{f \in P_T(s)} \widetilde{W}_T(f) \tag{14}$$

The weight computation algorithm integrates the idea of adaptive weighting with multiple factors, including indirect call confidence, depth information, distance metrics, and target weights, thereby realizing the four guiding strategies.

To visually demonstrate the effectiveness of the seed weighting algorithm, we use FreeImage as an example and present a runtime weight distribution chart in Fig. 7. We set $jpeg\_read\_exif\_dir$ as the target function. Starting from $FreeImage\_LoadFromMemory$, we visualize a portion of the CG after executing with an empty seed for ten minutes. In this graph, both $FreeImage\_LoadFromHandle$ and $FreeImage\_GetFIFCount$ contain indirect calls. The function $FreeImage\_LoadFromHandle$ can indirectly trigger 31 different Load functions. For better readability, we use $Load01\_10\ Avg$ to denote the average weight of the first 10 Load functions, and $Load12\_31\ Avg$ to denote the average of the remaining 20. Among them, $Load11$ is the function on the actual vulnerability-triggering path.

As shown in the figure, function nodes closer to the TF have significantly higher weights, while nodes unrelated to the TF exhibit low weights, often close to zero. We will conduct a more quantitative evaluation of the guiding effect of this algorithm in Section 5.4.

4.5. Seed sorting algorithm

To accelerate the discovery of target functions, the seed input queue is implemented as a priority queue. A seed's priority is determined collectively by its weight and execution count. The seed weight reflects its potential to generate new coverage or trigger new indirect calls through mutation. However, as the execution count of a seed increases, the number of times it has been mutated also increases, gradually reducing its potential to produce new coverage. Consequently, once a seed has been sufficiently executed, its priority will approach zero.

For any seed that has been executed at least once, its priority is calculated as Eq. (15), where $execCount(s)$ denotes the number of times seed s has been executed:

$$prio_n(s) = \frac{W_S(s)}{\sqrt{execCount(s)}} \tag{15}$$

The seed priority is calculated as shown in Eq. (16), where $seedWith Coverage(s)$ indicates whether seed s has discovered new coverage. Seeds that trigger new coverage are prioritized for exploration. $MaxSP$ and $MinSP$ denote the maximum and minimum priority values among all seeds, respectively.

$$prio(s) = \begin{cases} 1 & \text{if } seedWithCoverage(s) \\ \frac{prio_n(s) - MinSP}{MaxSP - MinSP} & \text{else} \end{cases} \tag{16}$$

Based on this formula, we propose a seed priority scheduling algorithm, as illustrated in Algorithm 2. We set up two priority queues for seeds—the high-priority queue stores seeds with high priority or seeds that have found new coverage. Moreover, the low-priority queue holds seeds with lower priority. It is essential to note that any new seed that fails to uncover new coverage will be discarded immediately.

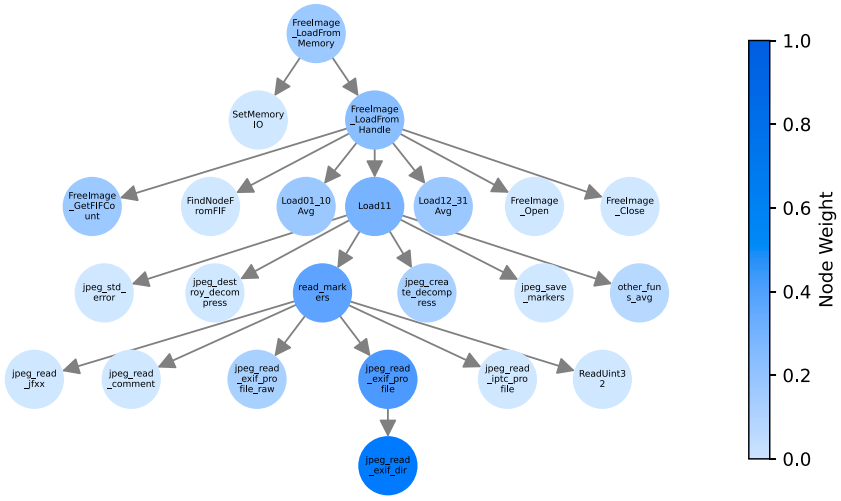

Fig. 7. Runtime weight distribution chart. The node color indicates the corresponding weight—the darker the color, the higher the weight.

Algorithm 2 Seed Prioritizing Algorithm

Input: $highPriorQueue$, $lowPriorQueue$, $seed$
Output: $highPriorQueue$, $lowPriorQueue$
if seedWithCoverage($seed$) **then**
  addQueue($highPriorQueue, seed$)
  return $highPriorQueue, lowPriorQueue$
end if
$seed ExecTimes \leftarrow$ execTimes($seed$)
$seed Priority \leftarrow$ calculatePriority($seed, seed ExecTimes$)
if $seed Priority > 0.8$ **then**
  addQueue($highPriorQueue, seed$)
else
  addQueue($lowPriorQueue, seed$)
end if
return $highPriorQueue, lowPriorQueue$

4.6. Power scheduling

During fuzzing, the fuzzer mutates seeds to generate new ones. The power scheduling algorithm addresses the allocation of mutation opportunities, ensuring that seeds closer to the target have more mutation attempts, thereby increasing the probability of reaching the target. Our primary focus is not on optimizing the power scheduling algorithm. Hence, we adopt the simulated annealing algorithm from AFLGo (Böhme et al., 2017) to minimize the influence of extraneous factors.

The temperature equation (Eq. (17)) is the same as AFLGo (Böhme et al., 2017), where T_{exp} denotes the global temperature, t represents the current running time of the algorithm, and t_x indicates the expected time to enter the development phase, set to 1 h.

$$T_{exp} = 20^{-\frac{t}{t_x}} \tag{17}$$

We employ an exponential cooling schedule to define the annealing-based power schedule (APS). Given the seed s, the APS assigns energy P as Eq. (18), where $\widetilde{W}_S(s)$ represents min–max normalization for seed weights.

$$P(s, T_{exp}) = (1 - \widetilde{W}_S(s)) \times (1 - T_{exp}) + 0.5T_{exp} \tag{18}$$

In combination with AFL's energy scheduling method, the seed energy is calculated as shown in Eq. (19). where s represents an arbitrary seed and $p_{afl}(s)$ is the energy assigned by AFL to seed s. A seed positioned closer to the target location receives higher energy allocation, up to a maximum of 32 times the original energy p_{afl}.

$$Power(s, T_{exp}) = p_{afl}(s) \times 2^{10 \times P(s, T_{exp}) - 5} \tag{19}$$

4.7. Optimization strategies

The graph analysis component of DynamicFuzz has considerable overhead. However, we have integrated several optimizations to ensure its practical performance on large-scale programs.

(1) Incremental and On-Demand Updates. DynamicFuzz's core data structures, such as the CG and node weights, are not recalculated from scratch. The update logic is triggered only when an ICE's confidence changes, and the update's scope is confined to the local subgraph affected by the ICE.

(2) Reachability-Based Pruning. During the static analysis phase, we pre-calculate the reachability from all ICEs to the TFs. ICEs that cannot reach any TF are marked as "ignored" and are subsequently disregarded in confidence updates and path calculations, thereby significantly reducing the size of the graph that requires dynamic maintenance. These ignored ICEs are only considered for updates if no reachable path exists from an entry function to a target function within the CG.

(3) Pre-construction of Function Islands. Since the function nodes within a function island are connected solely by direct calls, all potential function islands can be pre-constructed during the initialization of the CG. At runtime, an island is constructed only when the execution of a seed discovers or verifies a new ICE. At this point, the corresponding island can be retrieved directly from the pre-built list without additional path analysis. Therefore, this island construction process incurs no extra performance overhead.

(4) Dual-process Architecture. By default, the fuzzer uses native AFL instrumentation. When new coverage is discovered, DynamicFuzz activates a separate process that employs its own specific instrumentation to capture the seed path. This design ensures that the analysis overhead is incurred only when necessary, thereby minimizing the average impact on performance.

5. Implementation & evaluation

The DynamicFuzz implementation comprises a static analyzer and a dynamic fuzzer. The static analyzer utilizes the Clang opt tool and indirect call analysis tool SEA (Cheng et al., 2024) to generate the base CG, implementing code instrumentation via LLVM's pass mechanism, totaling approximately 600 lines of C/C++ code. The dynamic fuzzer

utilizes the Boost library for graph updates and seed weight calculations, comprising approximately 700 lines of C/C++ code. Seed priority queuing and scheduling algorithms are implemented based on AFL-2.52b, accounting for roughly 200 lines of C/C++ code. We evaluated DynamicFuzz through experiments to answer the following questions:

RQ1: How efficient is DynamicFuzz in reaching predefined targets?

RQ2: How efficient is DynamicFuzz in reproducing vulnerabilities?

RQ3: What is the impact of each component and guiding strategy on its performance?

RQ4: How does the choice of depth calculating method affect DynamicFuzz's guidance efficiency?

RQ5: How sensitive is DynamicFuzz's performance to the confidence decay parameter α?

RQ6: Can DynamicFuzz effectively perform real-world vulnerability mining?

5.1. Evaluation setup

5.1.1. Evaluation benchmarks

We use two datasets and a real program with potential vulnerabilities.

(1) FreeImage (Böhme et al., 2017) is an open-source project. Each vulnerability within FreeImage involves at least one indirect call, a scenario where conventional DGF tools can exhibit direction bias due to path computation being interfered with by these indirect calls. We randomly choose four vulnerabilities featuring buffer overflow as our test suite. This test suite addresses RQ2 and RQ6.

(2) The AFLGo Test Suite (Böhme et al., 2017) consists of programs containing n-day vulnerabilities utilized in the AFLGo's experiment. This test suite has been employed in various studies (Böhme et al., 2017; Chen et al., 2018) to assess DGF techniques. It was used to address RQ1, RQ2, RQ3, RQ4 and RQ5.

(3) UniBench (Li et al., 2021) offers a variety of real-world test programs widely used by current state-of-the-art fuzzers (Du et al., 2022; Lin et al., 2023). This test suite was used to address RQ1, RQ3, RQ4 and RQ5.

5.1.2. Baseline

Since DynamicFuzz builds on AFL, the power scheduling strategy and some configuration parameters are deliberately aligned with AFLGo; we compare DynamicFuzz with AFL and AFLGo to avoid potential errors due to implementation differences in the technology, as well as to test the effectiveness of our guiding method. Through priority decay, our algorithm ensures an equal guarantee of fair exploration of efficient paths, which is why we include FairFuzz in the comparison. This paper focuses on demonstrating the effectiveness of our guiding strategies in a setting with a high number of indirect calls, rather than pursuing advanced algorithmic fusion. To reduce experimental errors caused by multiple techniques and to highlight the focus of the paper, Windranger (Du et al., 2022), SelectFuzz (Luo et al.), WAFLGo (Xiang et al.), and other state-of-the-art fuzzers (Zou et al., 2023; Kim et al., 2023; He and Zhu, 2023; Rong et al., 2024; Xiang et al., 2024) in recent years are not included as comparisons, as their studies are in a compatible relationship with ours. DynamicFuzz can easily integrate the interesting ideas of these papers, which could further enhance performance.

5.1.3. Evaluation criteria

Two criteria are employed to assess the efficacy of various fuzzing techniques:

(1) Time-to-Target (TTT), measuring the duration required to produce the first seed that can reach the target site.

(2) Time-to-Exposure (TTE), quantifying the time taken to discover the vulnerability of the target site.

Table 2
TTT results on programs from UniBench and AFLGo test suite. The shortest μTTT is marked with an asterisk. For each target site, the statistically significant (p-value <0.05) value of $\hat{A}_{12}$ is bolded.

Prog	Targets	Tool	Hits	μTTT	Factor	$\hat{A}_{12}$
imginfo	jas_icc.c:1084	DynamicFuzz	10	*23 m 41 s	–	–
		AFL	10	2 h 49 m	7.1	**1**
		AFLGo	10	57 m 14 s	2.42	0.62
		FairFuzz	10	1 h 4 m	2.74	0.67
	jpc_cs.c:1497	DynamicFuzz	10	*25 m 25 s	–	–
		AFL	10	3 h 59 m	9.44	**1**
		AFLGo	10	1 h 10 m	3.14	0.66
		FairFuzz	10	34 m 49 s	1.37	0.56
	jas_image.c:298	DynamicFuzz	10	*4 m 34 s	–	–
		AFL	10	8 m 36 s	1.88	0.75
		AFLGo	10	4 m 59 s	1.09	0.68
		FairFuzz	10	19 m 24 s	4.24	**0.79**
objdump	objdump.c:1709	DynamicFuzz	10	*26 m 21 s	–	–
		AFL	10	41 m 49 s	1.63	0.67
		AFLGo	10	43 m 48 s	1.71	**0.89**
		FairFuzz	10	35 m 26 s	1.38	0.66
	section.c:928	DynamicFuzz	10	*16 m 42 s	–	–
		AFL	9	2 h 32 m	9.1	**0.78**
		AFLGo	8	52 m	3.11	**0.89**
		FairFuzz	10	2 h 12 m	7.9	0.73
nm-new	hash.c:777	DynamicFuzz	10	*4 m 19 s	–	–
		AFL	10	38 m 21 s	8.86	**0.93**
		AFLGo	10	11 m 46 s	2.72	0.55
		FairFuzz	10	5 m 38 s	1.3	0.69
	coffcode.h:1187	DynamicFuzz	10	*19 m 20 s	–	–
		AFL	10	41 m 10 s	2.13	0.75
		AFLGo	10	39 m 18 s	2.03	0.65
		FairFuzz	10	49 m 57 s	2.58	0.5
mjs	mjs.c:13525	DynamicFuzz	10	*14 s	–	–
		AFL	10	28 s	1.95	**0.83**
		AFLGo	10	15 s	1.07	0.65
		FairFuzz	10	26 s	1.82	0.76
	mjs.c:12237	DynamicFuzz	10	*6 m 58 s	–	–
		AFL	10	1 h 21 s	8.67	**0.86**
		AFLGo	10	39 m 25 s	5.67	**0.86**
		FairFuzz	10	18 m 32 s	2.66	**0.82**

5.1.4. Experiment setting

We conducted experiments on a machine equipped with an Intel(R) Core(TM) i9-10980XE CPU @ 3.00 GHz, featuring 32 cores, and running Ubuntu 22.04 as the operating system. We launched 10 fuzzer instances simultaneously for each TTT test and 20 fuzzer instances for each TTE test, with a 24-hour time limit for each experiment. Confidence decay parameter α is set to 1.0001. We utilized the Mann–Whitney U test (p-value) to assess the statistical significance of experimental outcomes. We employed the Vargha-Delaney statistic ($\hat{A}_{12}$) to determine the likelihood of one technique outperforming another.

5.2. Target site reaching capability

Reaching the target site stands as a crucial aspect of DGF. To answer RQ1, we randomly selected and tested a total of 8 target sites in UniBench and AFLGo Test Suite that contain indirect calls as test subjects. The results of the experiments are listed in Table 2. To visualize the results, we use bar charts to visualize the results of 80 experiments. In Fig. 8, the x-axis represents the target experiment ID, and the y-axis represents the TTT of fuzzers. Fig. 8 clearly shows that DynamicFuzz is faster than other fuzzing tools in most of the experiments. As shown in Table 2, DynamicFuzz exhibits the shortest μTTT and the highest hit rate across all targets. In particular, DynamicFuzz is faster than AFL, AFLGo, and FairFuzz in terms of average TTT by 5.64x, 3.01x, and 2.89x, respectively. Comparing the $\hat{A}_{12}$ metrics with other tools, DynamicFuzz consistently outperforms other tools. Hence, we are confident enough to believe that DynamicFuzz has a better ability to reach the target site than other tools.

While efficiently reaching predefined code locations is a fundamental measure of a DGF's capability, its ultimate value lies in exposing vulnerabilities. We will now shift our evaluation from reachability to crash discovery to answer RQ2.

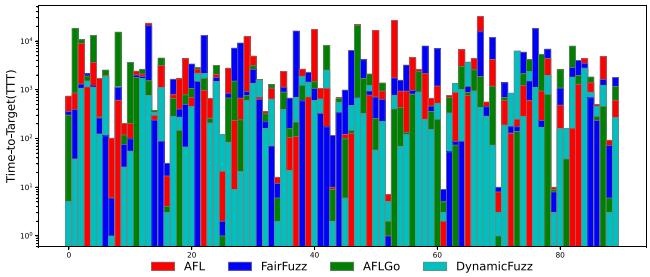

Fig. 8. Results of 90 TTT experiments by AFL, FairFuzz, AFLGo, DynamicFuzz in UniBench and AFLGo Test Suite at 9 target sites.

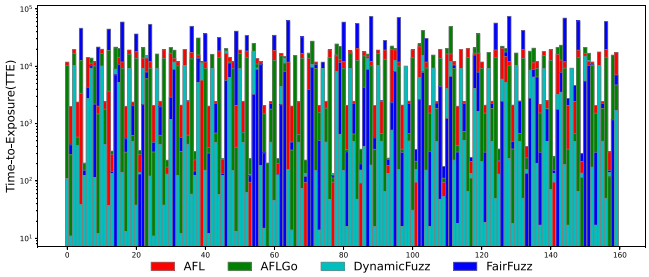

Fig. 9. Results of 160 TTE experiments by AFL, FairFuzz, AFLGo, DynamicFuzz at 8 CVEs in FreeImage and AFLGo Test Suite.

Table 3
TTE results. The shortest μTTE is marked with an asterisk. The shortest μTTE is marked with an asterisk. For each CVE, the statistically significant (p-value <0.05) value of $\hat{A}_{12}$ is bolded.

Test suite	CVE-ID	Tool	Hits	μTTE	Factor	$\hat{A}_{12}$
FreeImage Test Suite	2021–40266	DynamicFuzz	20	*2 m47 s	–	–
		AFL	18	3 h18 m	71.5	**1**
		AFLGo	20	2 h39 m	57.28	**1**
		FairFuzz	16	11 h8 m	241.62	**1**
	2023–47993	DynamicFuzz	20	*15s	–	–
		AFL	20	34 m23 s	138.94	**1**
		AFLGo	20	12 m39 s	51.16	**1**
		FairFuzz	20	1 h16 s	243.6	**1**
	2023–47994	DynamicFuzz	18	*2 h39 m	–	–
		AFL	10	5 h18 m	2	**1**
		AFLGo	14	4 h15 m	1.6	**0.9286**
		FairFuzz	11	3 h17 m	1.2	**1**
	2023–47996	DynamicFuzz	20	*7 m54 s	–	–
		AFL	20	40 m37 s	5.13	**1**
		AFLGo	20	17 m46 s	2.24	**1**
		FairFuzz	16	40 m6 s	5.07	**1**
AFLGo Test Suite	2017–16883	DynamicFuzz	20	*2 m13 s	–	–
		AFL	20	3 m12 s	1.44	**0.79**
		AFLGo	20	3 m54 s	1.75	**0.84**
		FairFuzz	20	2 m56 s	1.32	0.48
	2018–8807	DynamicFuzz	19	*2 h28 m	–	–
		AFL	20	3 h44 m	1.51	0.2275
		AFLGo	20	5 h38 m	2.28	0.48
		FairFuzz	20	2 h38 m	1.06	0.37
	2018–8962	DynamicFuzz	19	1 h58 m	–	–
		AFL	19	4 h47 m	2.44	**0.83**
		AFLGo	19	5 h49 m	2.96	**0.92**
		FairFuzz	20	*1 h57 m	0.99	0.53
	2017–5969	DynamicFuzz	20	*52 s	–	–
		AFL	20	4 h50 m	335.7	**1**
		AFLGo	20	3 h51 m	267.7	**1**
		FairFuzz	15	11 h28 m	794.7	**1**

5.3. Crash exposure capability

Replicating bugs or crashes is a significant application of DGF. This experiment aims to compare DynamicFuzz's crash exposure capabilities with those of other fuzzers, encompassing known crashes and our newly discovered ones.

5.3.1. FreeImage test suite

FreeImage contains a large number of indirect calls, making it ideal for evaluating our method on programs with indirect calls. The time required to penetrate the magic number constraint exhibits significant randomness due to the randomness of the seed variants. Our primary focus is to evaluate the directionality of DynamicFuzz on programs with indirect calls; however, such randomness weakens the validity of the experimental results. To mitigate this effect, we randomly selected small seed images in different formats to minimize the effect of the magic number while maintaining the validity of the experiment.

5.3.2. AFLGo test suite

We compared the performance of DynamicFuzz with other tools on the AFLGo Test Suite (Böhme et al., 2017). Specifically, we chose to evaluate libming 0.4.8 and libxml2 2.9.2, which contain a total of 4 CVEs. The default seed provided in the AFLGo Test Suite was used as the initial seed in the experiments.

5.3.3. Experimental results

Table 3 shows the results of the experiments, and Fig. 9 visualizes the 160 experiments, with the x-axis representing the vulnerability experiment IDs and the y-axis representing the exposure time (TTE). The data shows that (1) DynamicFuzz achieves the shortest TTE and the highest number of hits on almost all CVEs, which is a significant improvement in vulnerability exposure time compared to other tools. (2) For CVEs that are easy to expose, DynamicFuzz achieves accelerated performance by prioritizing the execution of the more valuable seeds. (3) For challenging CVEs (μTTE> 1 h), DynamicFuzz is also significantly faster than other tools by 1.2 to 2.9 times. Overall, DynamicFuzz demonstrates superior performance compared to other tools, averaging 69.8x, 48.37x, and 161.20x faster than AFL, AFLGo, and FairFuzz, respectively.

Having demonstrated that DynamicFuzz is effective at both reaching targets and exposing crashes, we now turn to RQ3 to analyze why it is effective. The following ablation studies systematically evaluate the impact of each core component and guidance strategy.

5.4. Impact of different components

To answer RQ3 and evaluate the impact of DynamicFuzz's different components and strategies, we conducted two separate ablation studies. Additionally, we targeted $mjs.c$: 13525 in the mjs program to test the exploration progress along different target-reachable paths during the fuzzing process, thereby validating the effectiveness of our seed prioritization strategy.

To investigate the effects of the seed priority queue module and the seed guidance strategy on DynamicFuzz, we individually disabled each component and measured the TTT results. The experimental results are presented in Table 4. SSA indicate Seed Sorting Algorithm, GS indicate Guiding Strategies. As shown in Table 4, disabling either component increases TTT, indicating that each component has a significant impact on DynamicFuzz's performance. The results show that the seed sorting algorithm provided an average speed-up of 26.89%. However, the fundamental reason for DynamicFuzz's superior results lies in the guidance strategy proposed in this paper. As evidenced by the experimental results, the guidance algorithm accelerated the TTT by 289.83%.

To evaluate the contribution of the four strategies proposed in this paper, we conducted an ablation study by sequentially disabling the Target Function Selection (TFS), Function Island Prioritization (FIP), High-Confidence Path Prioritization (HCP), and Deep Indirect Call Prioritization (DICP) strategies, measuring their impact on the μTTT. To more clearly demonstrate the effect of the DICP strategy, we conducted two experiments with the DICP strategy disabled. In the first experiment, we removed all pre-built ICEs (R-ICE), such that DynamicFuzz could not identify any valid path to the TF. In the second experiment, we used SEA to construct the CG typically (C-ICE). The results are presented in Table 5. As all experiments reached the target within 24 h, the 'Runs' column is omitted from the table.

Table 4
Ablation experiment of DynamicFuzz's components.

Prog	Targets	Default		No SSA		No GS	
		Runs	μTTT	Runs	μTTT	Runs	μTTT
imginfo	jas_icc.c:1084	10	*23 m 41 s	10	28 m 33 s	10	1 h 22 m
	jpc_cs.c:1497	10	*25 m 25 s	10	30 m 37 s	8	3 h 18 m
	jas_image.c:298	10	*4 m 34 s	10	5 m 23 s	10	10 m 41 s
objdump	objdump.c:1709	10	*26 m 21 s	10	38 m 55 s	10	1 h 42 m
	section.c:928	10	*16 m 42 s	10	19 m 26 s	10	1 h 23 m
nm-new	hash.c:777	10	*4 m 19 s	10	5 m 3 s	10	12 m 40 s
	coffcode.h:1187	10	*19 m 20 s	10	23 m 3 s	10	56 m 10 s
mjs	mjs.c:13525	10	*14 s	10	32 s	10	49 s
	mjs.c:12237	10	*6 m 58 s	10	8 m 25 s	10	49 m 1 s
μTTT Inc		–		+26.89%		+289.83%	

Table 5
Ablation experiment of DynamicFuzz's guiding strategies.

Prog	Targets	Default	No TFS	No FIP	No HCP	No DICP (R-ICE)	No DICP (C-ICE)
imginfo	jas_icc.c:1084	*23 m 41 s	55 m 51 s	46 m 11 s	41 m 15 s	50 m 28 s	24 m 3 s
	jpc_cs.c:1497	*25 m 25 s	2 h 1 m	58 m 5 s	52 m 43 s	1 h 2 m	28 m 1 s
	jas_image.c:298	*4 m 34 s	4 m 55 s	4 m 39 s	5 m 18 s	9 m 49 s	4 m 37 s
objdump	objdump.c:1709	*26 m 21 s	42 m 10 s	47 m 3 s	35 m 20 s	49 m 41 s	28 m 40 s
	section.c:928	*16 m 42 s	45 m 38 s	30 m 20 s	51 m 48 s	41 m 3 s	18 m 5 s
nm-new	hash.c:777	*4 m 19 s	10 m 2 s	6 m 3 s	5 m 5 s	8 m 59 s	4 m 45 s
	coffcode.h:1187	*19 m 20 s	38 m 15 s	35 m 11 s	25 m 30 s	39 m 22 s	20 m 55 s
mjs	mjs.c:13525	*14 s	14 s	15 s	14 s	14 s	14 s
	mjs.c:12237	*6 m 58 s	17 m 4 s	12 m 55 s	10 m 22 s	14 m 10 s	7 m 35 s
μTTT Inc		–	162.73%	88.69%	78.40%	116.17%	7.33%

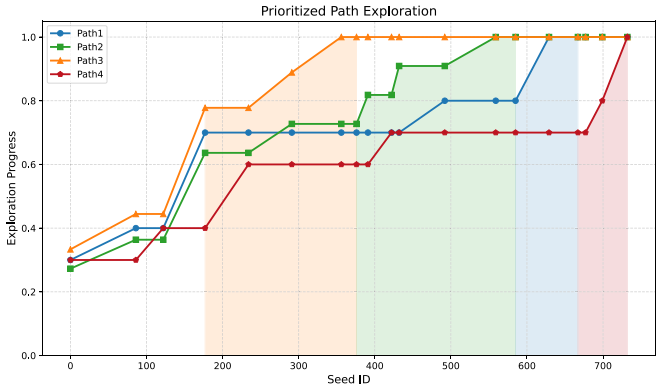

Fig. 10. Progress in the exploration of target-reachable paths during fuzzing.

Disabling any single strategy, as shown in Table 5, led to a decline in performance, indicating that all components contribute positively to the overall effectiveness of DynamicFuzz. The results demonstrate that the TFS strategy achieves an average speedup of 162.73%. This significant improvement is due to TFS being the foundational strategy for DynamicFuzz's directed guidance; consequently, disabling it renders other strategies ineffective. The FIP strategy provides an average speedup of 88.69%, indicating that our insights regarding function islands significantly enhance the guidance efficiency of DGF. The HCP strategy achieved an average speedup of 78.40%, indicating that it effectively filters out unreliable ICEs and guides DGF to prioritize reliable paths. When no path to the TF exists in the CG, the DICP strategy provides a general directional trend, achieving an average speedup of 116.17%. However, when paths to the TF were generally available in the CG, the strategy had no significant effect, resulting in a speedup of only 7.33%.

To assess the effectiveness of the seed prioritization strategy proposed in this paper, we examined the progress of exploration for various reachable paths targeting mjs.c:13525 within the mjs program. The results of our experiments are illustrated in Fig. 10. The horizontal axis represents the seed index. In contrast, the vertical axis indicates the progress of path exploration for different reachable paths. The shaded area between each path's line and the horizontal axis illustrates the time during which that path was preferentially explored.

In the initial phase of fuzzing, the program coverage is low, and DynamicFuzz does not show a tendency to prioritize any specific path. However, at seed ID 177, the nodes in Path 3 are closer to the TF. As a result, seeds associated with Path 3 are scheduled preferentially, allowing for rapid and complete exploration of this path. Even after Path 3 is fully explored, it remains a priority to identify potential vulnerabilities. As the execution count for Path 3 increases, the priority of its related seeds gradually diminishes. DynamicFuzz then reallocates computational resources to the next highest priority path, Path 2, followed by Paths 1 and 4. The experimental results demonstrate that our seed scheduling algorithm effectively balances the "exploitation" of high-value seeds with the necessary "exploration" of diverse paths, successfully preventing the phenomenon of starvation

The ablation study highlighted the importance of our guidance strategy. To further validate our specific design choices within this strategy, we now analyze the sensitivity of two key heuristics: the depth calculation method (RQ4) and the confidence decay rate (RQ5).

5.5. Impact of depth computing strategy

To address RQ4 and validate our depth calculation method, we compared our default weighted average strategy with five common depth calculation strategies: the minimum (Min), maximum (Max), arithmetic mean (Mean), median (Median), and mode (Mode) of the depths of the parent nodes. Fig. 11 illustrates the proportion of each strategy's μTTT to the total μTTT of all strategies.

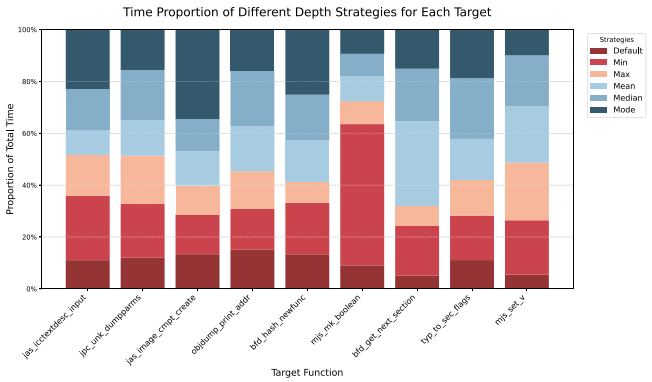

Fig. 11. Proportion of the value of μTTT for each strategy relative to the total value of μTTT across all strategies when different depth calculation methods are used.

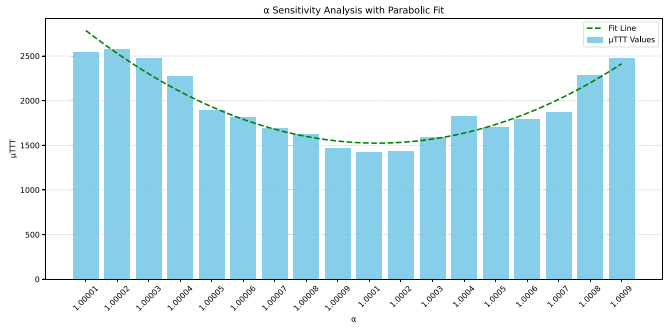

Fig. 12. The sensitivity of DynamicFuzz to the confidence decay parameter α.

The results indicate that our proposed weighted average method demonstrates highly competitive performance for the majority of targets, providing the best balance between speed and stability. This outcome empirically supports the design choice we discussed in Section 4.3.

5.6. Impact of confidence decay rate

To address RQ5, we evaluated the sensitivity of DynamicFuzz to the confidence decay parameter α (as described in Eq. (6)). We used the function $jas\_icc.c : 1084$ from the $imginfo$ as our target. We varied α within the range of 1.00001 to 1.0009 and recorded the μTTT. As illustrated in Fig. 12, there is a noticeable dip in the performance curve. Choosing α values that are excessively small or large could lead to confidence decaying too slowly or too quickly. This, in turn, affects the efficiency of pruning incorrect ICEs. The selected value for our experiments, 1.0001, is located within a near-optimal region of this curve, which supports our choice.

5.7. Answers to research questions

With the above experimental results, we can answer the research questions.

RQ1: Results in Table 2 demonstrate that DynamicFuzz has the capability to reach the target more rapidly than alternative tools.

RQ2: DynamicFuzz demonstrates strong performance in replicating target vulnerabilities and identifying crashes. Table 3 illustrates its exceptional capabilities, particularly in programs with a higher occurrence of indirect calls, such as FreeImage.

RQ3: The analysis of Table 4 , Table 5 and Fig. 10 reveals the impact of DynamicFuzz's components, dynamic guiding strategies and seed prioritization queue on the guiding effectiveness, which proves the effectiveness of the dynamic guiding strategies proposed in this paper.

RQ4: The analysis of Fig. 11 demonstrates the effectiveness and rationality of DynamicFuzz's depth calculation strategy.

RQ5: The analysis of Fig. 12 reveals the best choice for the confidence decay parameter α.

RQ6: The experiments conducted on FreeImage provide compelling evidence that DynamicFuzz proficiently exposes vulnerabilities in real-world scenarios.

6. Discussion

DynamicFuzz enhances the DGF by employing three methods: indirect call confidence, four guiding strategies, and a priority queue. Specifically, the indirect call confidence adjusts the edge weight in the CG based on the runtime behavior of the indirect call edge. As the confidence of a false reporting indirect call edge decreases, its misleading effect on seed guidance is reduced. The four guiding strategies integrate the function island, depth information, and indirect call confidence to optimize the seed guiding method, implicitly filtering out seeds with low value. The design of the priority queue ensures that high-value seeds are executed first. DynamicFuzz ensures that high-priority seeds are executed several times, and then their priority gradually decreases to avoid starving other seeds. In seed scheduling, we deliberately retain the scheduling strategy of AFLGo to validate the effectiveness of the dynamic guidance algorithm, thereby excluding any influence on the experimental results from variations in the seed scheduling algorithm. Experimental results demonstrate that DynamicFuzz outperforms other DGF methods in reaching target sites and exposing vulnerabilities, particularly in programs with indirect calls. Thus, the effectiveness of the proposed call graph construction and guiding strategies can be affirmed.

7. Threats to validity

Internal threats originate from three aspects: (1) In the various components of DynamicFuzz, we employ heuristic parameters based on empirical settings, and the values of these parameters may affect the performance of DynamicFuzz. Preliminary experiments indicate that the current parameter setting is reasonable and applicable for most testing scenarios. In future work, we plan to conduct more systematic studies to investigate the effects of these parameters and determine the optimal configuration. (2) To emphasize our guiding strategies and reduce the randomness induced by the *magic number*, we did not use empty seeds as the initial seeds in our experimental evaluation. Nevertheless, this difference does not significantly affect our experimental conclusions. To mitigate this threat, we performed repeated experiments with multiple seeds of smaller size and lower information, ensuring that all tools in the comparative experiments utilized the same seed corpus. (3) The third internal threat involves treating all indirect calls uniformly. Our current confidence model does not differentiate between various types of indirect calls, such as C++ virtual function table calls or function pointers used for callbacks. Developing a more refined model that incorporates this semantic or contextual information could improve the precision of our guidance. This represents a promising direction for future research, building on the general confidence framework proposed in this paper.

External threats primarily arise from the experimental setup. To reduce the randomness in the fuzzing process, we repeated our experiments 20 times. To minimize the impact of differing implementations on our dynamic strategies, we selected an implementation based on AFL, since DynamicFuzz is also implemented on AFL. Additionally, for aspects outside our focus, we maintained consistency with AFLGo.

8. Conclusions

In this paper, we propose a novel directed greybox fuzzer, DynamicFuzz. We address two technical challenges inherent in DGF. First, we mitigate the influence of indirect call omissions and misjudgments on seed guidance. Second, we conduct effective fuzzing on an unreliable call graph. Our approach dynamically constructs a call graph by continuously updating the fuzzing targets and adaptively adjusting our guidance based on four guiding strategies. To this end, we introduce the concept of function islands to distinguish the value of different indirect calls, and we improve the design of the priority queue while ensuring execution efficiency. Experimental results demonstrate that our method significantly optimizes fuzzing for binary programs with numerous indirect calls and outperforms state-of-the-art greybox fuzzers. Moreover, DynamicFuzz detected a total of 8 undisclosed vulnerabilities across various open-source libraries and these vulnerabilities have been assigned CVE numbers, which validates its effectiveness in vulnerability discovery.

CRediT authorship contribution statement

Hao Jiang: Writing – review & editing, Writing – original draft, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Kang Wang:** Validation, Supervision, Software, Data curation. **Yujie Yang:** Writing – review & editing, Validation, Supervision, Investigation. **Shan Zhong:** Writing – review & editing, Validation, Supervision, Software. **Shuai Zhang:** Writing – review & editing, Validation, Investigation. **Chengjie Liu:** Writing – review & editing, Supervision. **Xiarun Chen:** Writing – review & editing, Validation, Supervision. **Weiping Wen:** Writing – review & editing, Supervision, Resources, Project administration.

Declaration of competing interest

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

Data availability

Data will be made available on request.

References

Andersen, L.O., 1994. Program Analysis and Specialization for the C Programming Language. Technical report, University of Copenhagen.

Bai, W., Wu, K., Wu, Q., Lu, K., 2023. Guiding directed fuzzing with feasibility. In: 2023 IEEE European Symposium on Security and Privacy Workshops. EuroS&PW, IEEE, pp. 42–49. http://dx.doi.org/10.1109/EuroSPW59978.2023.00010.

Blair, W., Mambretti, A., Arshad, S., Weissbacher, M., Robertson, W., Kirda, E., Egele, M., 2020. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. http://dx.doi.org/10.48550/arXiv.2002.03416, arXiv preprint arXiv:2002.03416.

Böhme, M., Pham, V.-T., Nguyen, M.-D., Roychoudhury, A., 2017. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2329–2344. http://dx.doi.org/10.1145/3133956.3134020.

Böhme, M., Pham, V.-T., Roychoudhury, A., 2016. Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1032–1043. http://dx.doi.org/10.1145/2976749.2978428.

Cha, S.K., Woo, M., Brumley, D., 2015. Program-adaptive mutational fuzzing. In: 2015 IEEE Symposium on Security and Privacy. IEEE, pp. 725–741. http://dx.doi.org/10.1109/sp.2015.50.

Chen, P., Chen, H., 2018. Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 711–725. http://dx.doi.org/10.1109/SP.2018.00046.

Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H., Liu, Y., 2020a. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In: 29th USENIX Security Symposium. USENIX Security 20, pp. 2325–2342. http://dx.doi.org/10.48550/arXiv.2007.15943.

Chen, Y., Li, P., Xu, J., Guo, S., Zhou, R., Zhang, Y., Wei, T., Lu, L., 2020b. Savior: Towards bug-driven hybrid testing. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1580–1596. http://dx.doi.org/10.48550/arXiv.1906.07327.

Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., Liu, Y., 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 2095–2108. http://dx.doi.org/10.1145/3243734.3243849.

Cheng, B., Zhang, C., Wang, K., Shi, L., Liu, Y., Wang, H., Guo, Y., Li, D., Chen, X., 2024. Semantic-enhanced indirect call analysis with large language models. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. pp. 430–442.

Demetrescu, C., Italiano, G.F., 2005. Trade-offs for fully dynamic transitive closure on dags: breaking through the o(n2) barrier. J. ACM 52 (2), 147–156.

Du, Z., Li, Y., Liu, Y., Mao, B., 2022. WindRanger: a directed greybox fuzzer driven by deviation basic blocks. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2440–2451. http://dx.doi.org/10.1145/3510003.3510197.

Godefroid, P., Klarlund, N., Sen, K., 2005. DART: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. http://dx.doi.org/10.1145/1065010.1065036.

Godefroid, P., Levin, M.Y., Molnar, D.A., et al., 2008. Automated whitebox fuzz testing. In: NDSS, vol. 8, pp. 151–166. http://dx.doi.org/10.1007/978-3-642-02652-2_1.

He, Y., Zhu, Y., 2023. RLTG: Multi-targets directed greybox fuzzing. Plos One 18 (4), e0278138. http://dx.doi.org/10.1371/journal.pone.0278138.

Huang, H., Guo, Y., Shi, Q., Yao, P., Wu, R., Zhang, C., 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In: 2022 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 36–50. http://dx.doi.org/10.1109/SP46214.2022.9833751.

Kim, T.E., Choi, J., Heo, K., Cha, S.K., 2023. DAFL: Directed grey-box fuzzing guided by data dependency. In: 32nd USENIX Security Symposium. USENIX Security 23, pp. 4931–4948. http://dx.doi.org/10.5281/zenodo.8031029.

Li, Y., Ji, S., Chen, Y., Liang, S., Lee, W.-H., Chen, Y., Lyu, C., Wu, C., Beyah, R., Cheng, P., et al., 2021. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In: 30th USENIX Security Symposium. USENIX Security 21, pp. 2777–2794. http://dx.doi.org/10.48550/arXiv.2010.01785.

Li, P., Meng, W., Zhang, C., 2024. SDFuzz: Target states driven directed fuzzing. In: 33rd USENIX Security Symposium. USENIX Security 24, pp. 2441–2457.

Li, Y., Xue, Y., Chen, H., Wu, X., Zhang, C., Xie, X., Wang, H., Liu, Y., 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 533–544. http://dx.doi.org/10.1145/3338906.3338975.

Liang, H., Zhang, Y., Yu, Y., Xie, Z., Jiang, L., 2019. Sequence coverage directed greybox fuzzing. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, IEEE Computer Society, pp. 249–259. http://dx.doi.org/10.1109/ICPC.2019.00044.

Lin, P., Wang, P., Zhou, X., Xie, W., Lu, K., Zhang, G., 2023. Hypergo: Probability-based directed hybrid fuzzing. http://dx.doi.org/10.48550/arXiv.2307.07815, arXiv preprint arXiv:2307.07815.

Lu, K., Hu, H., 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1867–1881.

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

Österlund, S., Razavi, K., Bos, H., Giuffrida, C., 2020. ParmeSan: Sanitizer-guided greybox fuzzing. In: 29th USENIX Security Symposium. USENIX Security 20, pp. 2289–2306.

Parygina, D., Mezhuev, T., Kuts, D., 2024. LibAFL-DiFuzz: Advanced architecture enabling directed fuzzing. arXiv preprint arXiv:2412.19143.

Pham, V.-T., Nguyen, M.-D., Ta, Q.-T., Murray, T., Rubinstein, B.I., 2021. Towards systematic and dynamic task allocation for collaborative parallel fuzzing. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1337–1341.

Rong, H., You, W., Wang, X., Mao, T., 2024. Toward unbiased multiple-target fuzzing with path diversity. In: 33rd USENIX Security Symposium. USENIX Security 24, pp. 2475–2492.

Wang, J., Kang, Y., Wu, C., Hu, Y., Sun, Y., Ren, J., Lai, Y., Xie, M., Zhang, C., Li, T., et al., 2024. OptFuzz: optimization path guided fuzzing for JavaScript jit compilers. In: Proceedings of the 2024 USENIX Security Symposium. SEC'24.

Wang, J., Song, C., Yin, H., 2021. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In: Network and Distributed System Security Symposium. http://dx.doi.org/10.14722/ndss.2021.24486.

Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., Sui, Y., 2020a. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 999–1010. http://dx.doi.org/10.1145/3377811.3380386.

Wang, P., Zhou, X., Yue, T., Lin, P., Liu, Y., Lu, K., 2020b. The progress, challenges, and perspectives of directed greybox fuzzing. Softw. Test. Verif. Reliab. e1869. http://dx.doi.org/10.1002/stvr.1869.

Wen, C., Wang, H., Li, Y., Qin, S., Liu, Y., Xu, Z., Chen, H., Xie, X., Pu, G., Liu, T., 2020. Memlock: Memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 765–777. http://dx.doi.org/10.1145/3377811.3380396.

Xiang, Y., Zhang, X., Liu, P., Ji, S., Liang, H., Xu, J., Wang, W., Critical code guided directed greybox fuzzing for commits. In: 33rd USENIX Security Symposium. USENIX Security 24, pp. 2459–2474.

Xiang, Y., Zhang, X., Liu, P., Ji, S., Liang, H., Xu, J., Wang, W., 2024. Critical code guided directed greybox fuzzing for commits. In: 33rd USENIX Security Symposium. USENIX Security 24, pp. 2459–2474.

Zhu, K., Lu, Y., Huang, H., Yu, L., Zhao, J., 2021. Constructing more complete control flow graphs utilizing directed gray-box fuzzing. Appl. Sci. 11 (3), 1351. http://dx.doi.org/10.3390/app11031351.

Zou, Y., Zou, W., Zhao, J., Zhong, N., Zhang, Y., Shi, J., Huo, W., 2023. PosFuzz: augmenting greybox fuzzing with effective position distribution. Cybersecurity 6 (1), http://dx.doi.org/10.1186/s42400-023-00143-2.