applied sciences

MDPI

Article

Constructing More Complete Control Flow Graphs Utilizing Directed Gray-Box Fuzzing

Kailong Zhu, Yuliang Lu *, Hui Huang, Lu Yu and Jiazhen Zhao

College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; zhukailong@nudt.edu.cn (K.Z.); hhui_123@163.com (H.H.); yulu@nudt.edu.cn (L.Y.); jiazhenzhao@nudt.edu.cn (J.Z.)
* Correspondence: cee401_nudt@126.com

Abstract: Control Flow Graphs (CFGs) provide fundamental data for many program analyses, such as malware analysis, vulnerability detection, code similarity analysis, etc. Existing techniques for constructing control flow graphs include static, dynamic, and hybrid analysis, which each having their own advantages and disadvantages. However, due to the difficulty of resolving indirect jump relations, the existing techniques are limited in completeness. In this paper, we propose a practical technique that applies static analysis and dynamic analysis to construct more complete control flow graphs. The main innovation of our approach is to adopt *directed gray-box fuzzing (DGF)* instead of *coverage-based gray-box fuzzing (CGF)* used in the existing approach to generate test cases that can exercise indirect jumps. We first employ a static analysis to construct the static CFGs without indirect jump relations. Then, we utilize directed gray-box fuzzing to generate test cases and resolve indirect jump relations by monitoring the execution traces of these test cases. Finally, we combine the static CFGs with indirect jump relations to construct more complete CFGs. In addition, we also propose an *iterative feedback mechanism* to further improve the completeness of CFGs. We have implemented our technique in a prototype and evaluated it through comparing with the existing approaches on eight benchmarks. The results show that our prototype can resolve more indirect jump relations and construct more complete CFGs than existing approaches.

Keywords: control flow graph; hybrid analysis; directed gray-box fuzzing; indirect jump relations

Citation: Zhu, K.; Lu, Y.; Huang, H.; Yu, L.; Zhao, J. Constructing More Complete Control Flow Graphs Utilizing Directed Gray-Box Fuzzing. *Appl. Sci.* **2021**, *11*, 1351. https://doi.org/10.3390/app11031351

Academic Editor: Ricardo Colomo-Palacios
Received: 12 January 2021
Accepted: 25 January 2021
Published: 2 February 2021

1. Introduction

A control flow graph (CFG) represents all paths of a program that might be traversed during execution and is a fundamental data structure in program analysis. In a CFG, nodes represent basic blocks of instructions and directed edges represent jumps in the control flow. The CFG lays foundation for many other program analysis techniques, such as data flow analysis [1,2], taint analysis [3,4], and symbolic execution [5–7]. The CFG is also widely applied in program verification [8,9], malware detection [10–13], code similarity analysis[14–16], and software vulnerability detection [17,18]. Therefore, utilizing appropriate approaches to construct complete and precise CFG is necessary.

However, *indirect jumps* bring challenges to constructing complete CFGs [19]. In C/C++ programs, a program's jump can be categorized as direct or indirect. A direct jump has a statically specified target which points to a single location in the program, whereas an indirect branch has a dynamically specified target which may point to any number of locations in the program. Indirect jumps are commonly used to realize dynamic program behaviors by implementing common programming constructs, such as virtual function calls and calls through function pointers. Although indirect jumps are common and useful, due to their dynamic nature, it is usually difficult to resolve the target of an indirect jump through static analysis. This leads to inherent challenges in constructing complete CFGs.

Current solutions for CFG construction fall into three categories: static analysis, dynamic analysis, and hybrid analysis. Static techniques [20–22] do not need to concrete

execute of target programs, and only need to analyze the code to understand the program structure. The approaches can traverse the whole program code and has the advantages of high coverage and low time cost. Therefore, static analysis tools for constructing CFGs, such as IDA Pro [20] for binary code and LLVM [22] for source code, are widely applied in various program analyses. However, static techniques have poor completeness because it is difficult to resolve indirect jump relations statically.

Dynamic techniques execute programs on a set of test cases and extract the control flow information from the execution traces. The approaches can resolve a certain of indirect jumps and discover precise control flow. However, the completeness of the CFG constructed by the approaches depends on the capability of the test cases to cover indirect jumps. In order to improve the coverage of test cases, Xu et al. [23] proposed to systematically force a program's execution to explore both branches of each conditional. However, because forced execution is a heavyweight analysis technique, the approach still has poor coverage in analysis of large-scale programs.

In recent years, hybrid techniques [24–27] have been proposed to resolve indirect jumps that cannot be handled by pure static analyses, and to improve the completeness of CFGs. Babic et al. [24] combined dynamic and static techniques to construct CFGs. They computed an underapproximation of the transition relation by resolving indirect jumps with a set of seed tests, and augmenting the computed relation with statically computed direct jumps. In their approach, the completeness of CFGs still depends on the coverage of the seed test cases. However, in their work, they only generated test cases randomly and did not study how to generate effective test cases. Zhu et al. [26] proposed using *coverage-based gray-box fuzzing (CGF)* to generate test cases to handle indirect jumps.

Gray-box fuzzing is a scalable and practical approach to software testing. It is widely applied to vulnerability detection and test case generation. Existing gray-box fuzzers usually use an evolutionary algorithm to generate test cases based on the feedback information from the execution. CGF is the most prevalent fuzzing scheme, which guides test case generation with the coverage information and aims to generate test cases which can achieve the maximum code coverage. CGF uses lightweight instrumentation to gain coverage information during runtime. Test cases exercising new path are added to the seed pool, and new test cases are generated by mutating the inputs in the seed pool. AFL is the state-of-the-art coverage-based gray-box fuzzer. It and its extensions have shown high practicability and scalability in vulnerability detection.

Zhu et al. [26] used CGF for the first time in CFG construction and resolved a certain of indirect jumps. Nevertheless, the purpose of CGF is to improve the coverage of the whole program, not the coverage of indirect jumps. The approach spends a lot of resources to test the code unrelated to indirect jumps. Therefore, it cannot exercise indirect jumps efficiently, which resulting in the constructed CFG is not enough completeness.

In summary, the hybrid analysis is a promising approach to construct more complete and precise CFGs. How to generate test cases that exercise more indirect jumps is the key of hybrid approaches. In the state-of-the-art approach to constructing CFGs, the test case generation still faces the following problems.

- **Undirected.** In order to resolve indirect jumps, the existing approaches use CGF to generate test cases. CGF seeks to increase the coverage of a seed tests for entire analyzed program rather than indirect jumps. However, stressing code unrelated to indirect jumps is a waste of resources in constructing CFGs. Therefore, CGF is considered to be undirected and unable to resolve indirect jumps efficiently in CFG construction.
- **Unsustainable.** The existing CGF uses coverage information obtained during runtime to guide test case generation. Many previous work showed that CGF always reaches a "stuck" state because it cannot obtain new information in the later stage. In the state, testing is unable to generate test cases to cover new code and to exercise new indirect jumps.

In this paper, we propose a novel hybrid approach combining dynamic and static analysis to resolve indirect jumps and construct more complete CFGs for C/C++ programs. The approach is motivated by the observation that only a few jumps in programs are indirect, and directed testing of indirect jumps code may resolve more indirect jumps than CGF used in existing approaches. Instead of CGF, our key insight is to use *directed gray-box fuzzing (DGF)* to efficiently generate test cases that exercise indirect jumps.

DGF is an improvement of traditional gray-box fuzzing proposed by Böhme et al. [28]. Its main idea is to focus on interesting parts of code, rather than to spend a lot of time on undirected exploration of the whole program like coverage-based fuzzing. Previous studies [28–34] have shown that DGF has good practicability in the vulnerability detection. When testing large-scale programs, it can generate test cases to exercise the given targets more effectively than the coverage-based fuzzing.

More specifically, we first use static analysis to resolve direct jumps and construct static inter-procedural CFGs. Then, taking indirect jump locations as targets, DGF is used to generate test cases that can exercise as many indirect jumps as possible. We run the analyzed program with the generated test cases and resolve the indirect jump targets. Finally, the results of static and dynamic analysis are combined to construct the CFG with indirect control-flow transitions.

Our other observation is that the combined CFGs contains new structure information of the program, which can be used to further optimize the DGF-based test case generation. Therefore, we propose an *iterative feedback mechanism* to continuously optimize the input generation. The CFG constructed through multiple iterations is taken as the final output.

In summary, the ability of our approach to construct more complete CFG mainly depends on the following improvements: (1) the DGF technique is used to generate test cases that exercise indirect jumps instead of CGF, which can resolve more indirect jumps than undirected test cases generation used in existing approaches, and (2) an iterative feedback mechanism is proposed to continuously optimize the input generation, which alleviates the unsustainability of the existing approaches and further resolve more indirect jumps.

On the basis of the techniques, we implement a prototype, dubbed by DGF-CFGConstructor, for constructing CFGs of programs and evaluate it on eight benchmarks. In the experiments, we investigate the effectiveness of our DGF-based test case generation and the iterative feedback mechanism. The results demonstrate that our approach can resolve more indirect jump relations and construct more complete CFGs than existing approaches. As a preliminary step, we also evaluate the application of our techniques in real-world program analysis. The results show that our techniques is useful in discovering vulnerabilities deeply hidden in the code.

The main contributions of this paper are summarized as follows:

- We proposed a novel hybrid approach to construct more complete CFGs by utilizing the DGF technique in test case generation.
- We designed an iterative feedback mechanism to make a virtuous cycle between CFG combination and test case generation. The mechanism further improves the completeness of constructed CFGs.
- We implemented a prototype for constructing CGFs and evaluated the effectiveness of our techniques through comparing with the existing approaches on eight benchmarks.

The remainder of this paper is organized as follows. Section 2 presents that indirect jumps is widely distributed in real-world programs, which brings challenges to the CFG construction. The overview of the proposed approach is described in Section 3. Section 4 describes the details of the four key steps in our method. The implementation details of our prototype are presented in Section 5. Section 6 evaluates our techniques. We summarize the related work in Section 7 and provide our conclusions in Section 8.

2. Motivation

Resolving indirect jumps is the main challenges to construct complete CFGs. In this section, we illustrate the wide distribution of indirect jumps by observing real-world

programs. Then, a simple example is given to explain the impact of indirect jumps on the completeness of CFGs and the harm to security analysis.

2.1. The Distribution of Indirect Jumps

In C/C++, an indirect jump occurs when a function pointer or a virtual function is used to call function. We investigated the distribution of indirect jumps in eight popular programs and show the results in Table 1. The first four columns denote the basic information of programs, including project names, their versions, description, and lines of code. The average size of these programs is 216k lines of code.

Columns # F and # C list the number of functions and the number of function calls in the programs, respectively. Column # I and R_i present the number of indirect jumps and the proportion of indirect jumps in all function calls. We can see that all eight programs have indirect jumps. These programs have an average of 782 indirect jumps, accounting for 2.92% of all function calls. The target addresses of this kind of jump are determined at runtime, and it is difficult to precisely resolve it by pure static analysis. This brings challenges to constructing complete CFGs.

Table 1. The distribution of indirect jumps in real programs.

Project	Version	Description	LoC	# F	# C	# I	R_i
libxml2	2.9.10	Xml parser	496 k	4087	41,365	1904	6.56%
JasPer	2.0.16	Picture handler	33 k	756	8010	32	0.40%
libming	0.4.8	Swf parser	96 k	1612	23,912	523	2.14%
FreeType	2.10.0	Library to render fonts	219 k	1769	17,337	659	3.66%
libpng	1.6.37	Png handler	97 k	1053	11,301	75	0.66%
ImageMagick	7.0.8.6	Picture handler	553 k	6068	133,021	372	0.28%
libjpeg-turbo	1.5.1	image codec	91 k	620	16,080	1469	8.37%
libtiff	4.1.0	Tiff processor	138 k	1127	17,408	220	1.25%
Average	-	-	216 k	2136	33,554	782	2.92%

2.2. Motivating Example

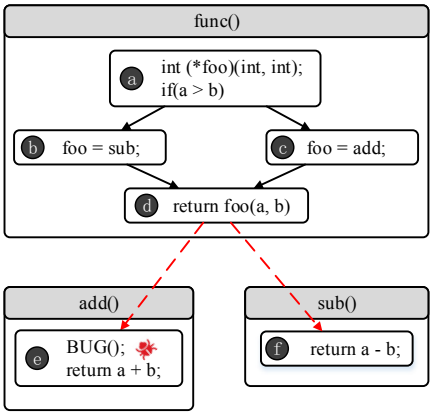
Figure 1 shows a simple example to explain the impact of indirect jumps, in which subfigures (a,b) give its source code and CFG, respectively. In function $func$, foo is a pointer to a function and may point to different targets when the input is different. If $a > b$, foo points to sub (line 11), otherwise it points to add (line 14). An indirect jump occurs when foo is used to call a function (line 16).

In Figure 1b, the dash lines, $d \rightarrow e$ and $d \rightarrow f$, represent the indirect jump relations. It is difficult to resolve the relations only by static analysis, which leads to incomplete CFGs. If a security analysis is based on the incomplete CFGs, the analysis results may be inaccurate. Assuming that there is a vulnerability in function add (line 5), we try to use flow sensitive analysis on CFGs to discover this vulnerability. If the CFG does not contain the indirect jump $d \rightarrow e$, the bug hidden in the function add cannot be found.

For this example, if we use the popular static analysis tool, such as IDA Pro, to construct the CFG, we will be unable to resolve the indirect jump addresses. The existing dynamic and hybrid approaches find it difficult to effectively generate test cases that exercise indirect jumps. The state-of-art hybrid approach uses a CGF-based approach to generate test cases. However, because this approach is undirected and unsustainable (in Section 1) in test case generation, it still cannot resolve indirect jumps effectively.

```
1  int sub(int a, int b){
2      return a - b;
3  }
4  int add(int a, int b){
5      BUG();
6      return a + b;
7  }
8  int func(int a, int b){
9      int (*foo)(int, int);
10     if(a > b){
11         foo = sub;
12     }
13     else{
14         foo = add;
15     }
16     return foo(a, b);
17 }
```

(**a**) Source code of the example



(**b**) CFG of the example

Figure 1. Example of indirect jumps. (**a**) is the source code of the example. (**b**) is the control flow graph (CFG) of the example.

3. Overview

In order to solve the aforementioned limitations in existing approaches, we propose a DGF-based hybrid approach to constructing CFGs. The approach aims to efficiently generate test cases that exercise indirect jumps and to construct more complete CFGs.

Figure 2 shows the overview of our proposed approach. The input is a program to be analyzed, which can be either source code or binary code. The output is the constructed inter-procedural control flow graph (iCFG) with indirect jumps. Our approach consists of three major components: static iCFG construction, DGF-based test case generation, and indirect jump monitoring.

Static iCFG construction. First, we use a static analysis to obtain the call graph (CG) of the program and CFGs of each function. There are some popular static tools that can be used to construct CG and functions' CFGs, such as LLVM [22] for source code and IDA Pro [20] for binary code. Then, we combine the CG and CFGs to construct the iCFG. We present more details in Section 4.1.

DGF-based test case generation. The main function of this component is to generate test cases that can exercise indirect jumps. First, a static analysis is used to search all indirect jump locations in the target program. Then, the indirect jump locations are taken as the targets to implement a distance-based DGF. DGF casts the reachability of target locations as optimization problem and minimizes the distance of the generated test cases to the targets. We collect the test cases that execute unique paths during the DGF and add them to the test case queue, which will be used in the following dynamic execution. The details of the test case generation are given in Section 4.2.

Indirect jump monitoring. The purpose of this component is to execute the target program using the generated test cases and to resolve the target addresses of indirect jumps through runtime monitoring. First, we instrument the target program at each of indirect jump locations. When an indirect jump is triggered, the instrumented program can record its target address. Then, all test cases generated by DGF are provided to the target program in turn. We can obtain all indirect jump relations triggered during the execution. More details of the component are shown in Section 4.3.

By combining the results of static iCFG construction and indirect jump monitoring, an iCFG with indirect jumps can be constructed.

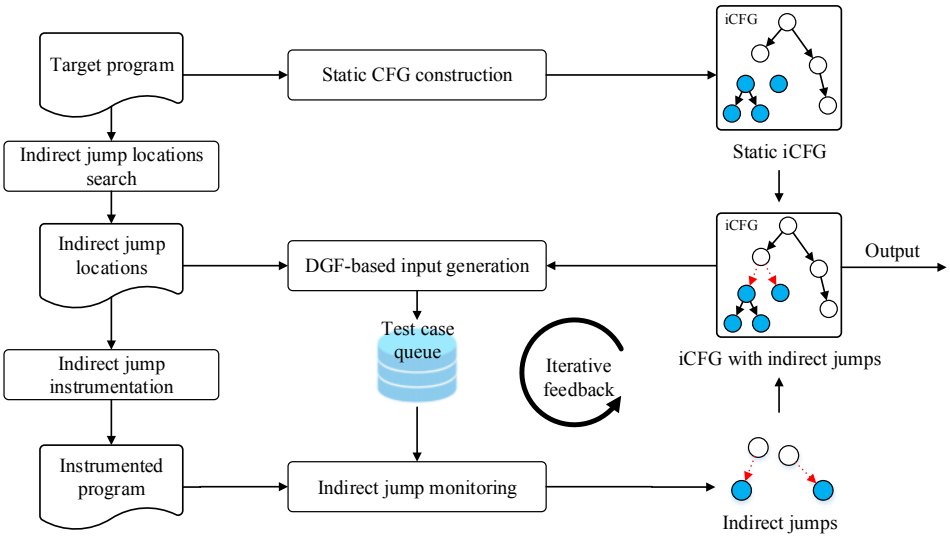


Figure 2. Overview of CFG construction based on directed gray-box fuzzing (DGF).

We observe that a new iCFG is constructed with the test cases generated by DGF. However, the distance calculation in DGF in turn depends on the iCFG. The new iCFG contains the new structure information of the program, which can be used to further direct the test case generation. Therefore, we propose an *iterative feedback mechanism* in CFG construction. The mechanism makes full use of the new information and further improves the completeness of CFGs. We present more details of the iterative feedback mechanism in Section 4.4.

The workflow of our approach is shown in Algorithm 1. At the beginning of the algorithm, a static analysis is used to analyze the target program p and to construct a static iCFG. Then, we search the locations of all indirect jump instructions in p, and instrument each of them to obtain the instrumented program p'. Next, the loop with iterative feedback starts at line 5. At the beginning of feedback loop, iCFG is set as static iCFG. Taking the locations of indirect jumps as the targets, DGF calculates the test case distance according to iCFG and generates test cases which are added to the test cases queue. Then, all instances in the test case queue are used to execute the instrumented program p', and the indirect jump relations are resolved through runtime monitoring. We update the iCFG by combining the new indirect jump edges with the original iCFG. The new iCFG are used as feedback to the DGF-based test case generation for next iteration, and lines 6–8 will be repeated. When a timeout is reached or the testing is aborted, the algorithm ends and outputs the iCFG updated in the last iteration.

Algorithm 1: The workflow of DGF-based CFG construction.

Input: target program p
Output: $iCFG$ of p with indirect jumps
1 $static\_iCFG$ = construct_static_CFG(p);
2 $iCFG$ = $static\_iCFG$;
3 $LOCs$ = search_indirect_jump(p);
4 p' = instrument(p, $LOCs$);
5 **while** *not (timeout or abort-signal)* **do**
6 $test\_case\_queue$ = DGF_generate_test_case(p, $LOCs$, $iCFG$);
7 $indirect\_jumps$ = execute_and_monitor(p', $test\_case\_queue$);
8 $iCFG$ = combine($iCFG$, $indirect\_jumps$);
9 **end**
10 output $iCFG$;

4. Methodology

This section elaborates the three key components and the iterative feedback mechanism in our approach.

4.1. Static CFG Construction

The main function of the component is to construct the iCFG for target program using static analyses. First, we construct CG and the function's CFGs. The CG of a program consists of the nodes representing functions and edges representing function calls. For each function, we construct a CFG in which nodes represent basic blocks and edges represent jumps among basic blocks. Some static analysis tools can provide the function to obtain CG and CFGs, such as LLVM's built-in APIs for source code and IDA Pro for binary code.

Based on the CG and CFG, we can construct iCFG by connecting all call-sites with the first basic block of the called functions. Figure 3 shows an example of the static iCFG construction. The gray basic blocks represent the basic blocks ending with call instructions. For call-site a, we can resolve the called function f_3. Then, the edge from call-site to the first basic block of called function, $a \rightarrow b$, is added to connect the CFGs of f_1 and f_2. Similarly, we can connect the CFGs of all functions according to the call relations to get the iCFG. It should be noted that the indirect jump edges are not included in the constructed iCFG because static tools cannot resolve them. In the following steps, we adopt dynamic analysis to resolve indirect jumps and augment the iCFG.

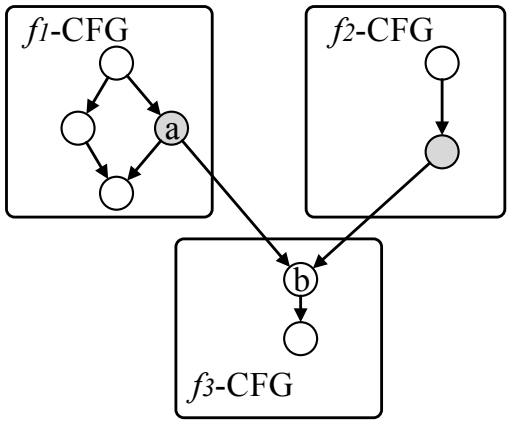


Figure 3. Example of the static inter-procedural control flow graph (iCFG) construction.

4.2. DGF-Based Test Case Generation

The test case generation is the core part in our approach, and its main function is to generate test cases that can exercise as many indirect jumps as possible. Aiming at the undirectedness of CGF that is used in the test case generation of existing CFG construction approach, we propose employing the distance-based DGF technique instead of CGF to generate test cases.

We introduce the DGF technique into the test case generation of CFG construction to resolve more indirect jumps. The main idea is to take the indirect jump locations as the targets of DGF and calculate the distance of each seed to the target code. According the distance, we evaluate the priority of seed in fuzzing evolution. Seeds closer to indirect jump locations gain higher priority. Therefore, the seed set evolves closer to indirect jump locations and may trigger more indirect jumps.

Figure 4 shows the workflow of the test case generation based on DGF. The test case generation contains two phases: distance calculation and fuzzing loop.

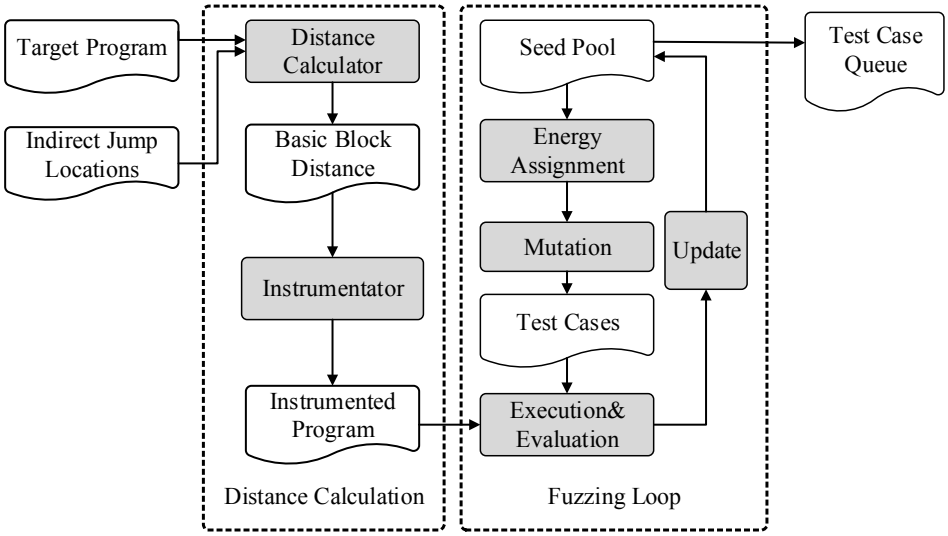


Figure 4. Input generation based on DGF.

4.2.1. Distance Calculation

The seed distance calculation is introduced from AFLGo [28], and it can determine the importance of the seed. The calculation is based on the constructed iCFG and consists of the following three steps.

(1) **Function-level distance on CG**. If a function contains indirect jump instructions, it is taken as target function. The set of all target functions is represented as F_t. We use $d_f(f_i, f_t)$ to represent the distance between any two functions on CG, i.e., the number of edges on the shortest path between the two functions. The function-level distance determines the distance between an arbitrary function to all target functions. Supposing a function f and target functions F_t are given, the function-level distance of f, represented as $d_f(f, F_t)$, is defined as the harmonic mean of the distance between the function f to all target functions as follows.

$$d_f(f, F_t) = \begin{cases} +\infty, & R(f, F_t) = \emptyset \\ \left[\sum_{f_t \in R(f,F_t)} d_f(f, f_t)^{-1}\right]^{-1}, & R(f, F_t) \neq \emptyset \end{cases} \quad (1)$$

where $R(f, F_t)$ is the set of functions that are members of F_t and are reachable from f.

(2) **Basic block-level distance on iCFG**. The basic blocks containing indirect jump instructions are called target basic blocks. The set of all target basic blocks is represents as B_t. We use $d_b(b_i, b_j)$ to represent distance between any two basic block. In a function, the distance is the number of edges on the shortest path between the two basic blocks on the CFG. The basic block-level distance determines the distance from an arbitrary basic block to target basic blocks, which is represented as $d_b(b_i, B_t)$. Based on the function-level distance, the basic block-level distance is calculated as follows.

$$d_b(b_i, B_t) = \begin{cases} 0, & if \quad b_i \in B_t \\ \alpha \cdot \min_{b_i \in N(f)}(d_f(f, F_t)), & if \quad b_i \in T \\ \left[\sum_{t \in T} (d_b(b_i, t) + d_b(t, B_t))^{-1}\right]^{-1}, & otherwise \end{cases} \quad (2)$$

where $N(f)$ is the set of all basic blocks in function f. α is a constant that represents the average length of functions in distance calculation. T represents a set of special basic blocks that we call *transfer basic blocks*. A transfer basic block should satisfy the following conditions: (a) the basic block ends with a call instruction and (b) the target function called by the call instruction can reach the target functions on CG.

(3) **Seed distance**. Based on the basic block-level distance, the distance from an arbitrary seed s to the target basic blocks is calculated and represented as $d(s, B_t)$. More specifically, the seed distance is defined as the average distance of all basic blocks on the seed execution trace, i.e.,

$$d(s, B_t) = \frac{\sum_{b \in \lambda(s)} d_b(b, B_t)}{|\lambda(s)|} \tag{3}$$

where $\lambda(s)$ is the set of basic blocks on the execution trace of the seed s.

An example is taken to describe the basic block-level distance calculation in Figure 5. In the example, a is the considered basic block and f is the target block. The gray basic blocks b and d are transfer basic blocks. According to the iCFG, there is a reachable path from a to f, i.e., $a \to b \to c \to d \to e \to f$. Therefore, the basic-block-level distance from a to the target block f is calculated as follows.

$$d_b(a, B_t) = d_b(a, b) + d_b(b, B_t) = d_b(a, b) + \alpha \cdot d_f(f_1, f_3) = 1 + \alpha \cdot 2.$$

The distances of function-level and basic-block-level (steps (1) and (2)) are calculated by static analysis. Only the seed distance calculation (step (3)) needs to be performed at runtime and instrumented into target programs.

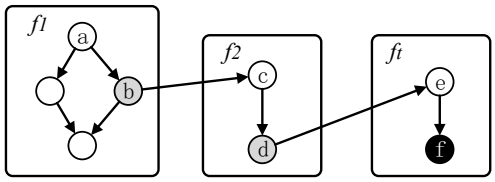


Figure 5. Example of basic block-level distance calculation.

4.2.2. Fuzzing Loop

Our fuzzing loop (the right part of Figure 4) is modified based on the classical gray-box fuzzing loop. The difference is that the purpose of the existing fuzzing is to find program vulnerabilities, while the purpose of our fuzzing is to generate test cases that exercise more indirect jumps. The workflow of our fuzzing loop is shown in Algorithm 2.

Algorithm 2: The fuzzing loop of test case generation.

Input: Initial seed set S_{in}
Output: The evolved seed set S_{out}

```
S = S_in;
while not (timeout or abort-signal) do
    s = Choose(S);
    p = AssignEnergy(s);
    for i from 1 to p do
        tcs = Mutate(s);
        info = Execute&Evaluate(tcs);
        Update(S, tcs, info);
    end
end
S_out = S;
output S_out;
```

The fuzzing loop takes an initial seed set S_{in} as input. In each loop, one seed s is chosen from S and assigned energy according to the seed distance. The energy represents the priority of seeds in the fuzzing evolution. The seeds closer to the targets (indirect jump locations) can obtain more energy and have more chances to generate test cases

tcs by mutating. Therefore, the mutation may generate test cases closer to the indirect jump locations in the next loop. Then, the program instrumented with seed distance calculation executes with the test cases *tcs* and evaluates each of them. The seed pool *S* is updated according to the results of the evaluation. By repeating the above steps, we can continuously update the seed pool to make the inputs closer to indirect jump locations. The fuzzing loop runs until a timeout or abort signal is received. Finally, all seeds in the evolved seed pool are output to the test case queue, which will be used to resolve indirect jump relations.

4.3. Indirect Jump Instrumentation and Monitoring

With the seed tests generated above, we execute the target program and monitor it at runtime to discover indirect jumps relations. First, in order to record indirect jump relations, the target program is instrumented at each indirect jump instruction and the entry of each function. We define a global value *Is_indirect_jump* to represent whether a function call is an indirect jump. At each indirect jump instruction, when the instruction is executed, *Is_indirect_jump* is assigned to *True* and the basic block that the indirect jump instruction belongs to is recorded as the source of the indirect jump, represented as b_{src}. At the entry of a function, we check the value of *Is_indirect_jump*. If it is *True*, the first basic block of this function is recorded as the target of the indirect jump, represented as b_{dst}. A indirect jump relation is represented as a pair (b_{src}, b_{dst}). Then, we provide all seed test cases generated in the previous stage (Section 4.2) to the instrumented program and resolve as many indirect jump relations as possible.

It is worth mentioning that we perform two different instrumentations for the original target program. One is to calculate the seed distance, while the other is to record the indirect jump relations. Theoretically, both of them can be performed in the process of fuzzing. However, inserting too many instructions in fuzzing may cause a lot of runtime overhead. Therefore, our approach separates the two instrumentations to minimize the overhead.

4.4. Iterative Feedback Mechanism

A new and more complete iCFG can be constructed by adding the discovered indirect jump edges (b_{src}, b_{dst}) to the old iCFG. We observed that the combined iCFG contains new structure information of the target program and can be used to optimize the DGF-based test case generation (Section 4.2). Therefore, we propose an iterative feedback mechanism to achieve a virtuous cycle between CFG combination and input generation and continuously improve the completeness of CFGs.

The key problem of iterative feedback mechanism is how to recalculate the seed distance on the new iCFG. If we simply use the approach in Section 4.2.1 to recalculate the distance, it will bring redundant time overhead. To minimize the overhead, we develop an incremental calculation to update the distance.

Figure 6 shows an example of updating the distance adopting the incremental calculation. The node *t* is the target basic block. The solid lines represent the jump edges in the old iCFG and the dash line $b \rightarrow c$ represents the newly discovered indirect jump edge. Because the new edge changes the structure of the iCFG, the basic block-level distance needs to be updated.

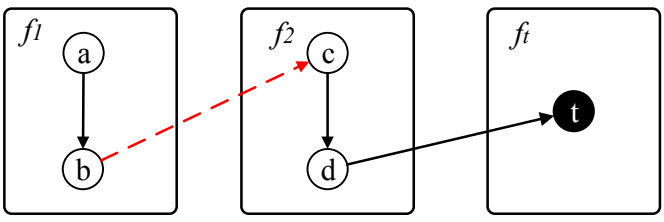


Figure 6. Example of updating the distance incrementally.

Through observation, it can be found that the new edge only affects the distance of the basic blocks before the edge on iCFG, but not the distance of the basic blocks after the edge. Therefore, we only need to calculate the distance of the precursors of the source basic block of the edge. In the example, the distances of a and b need to be recalculated, while the distances of c and d do not. The updated results of the example are shown in Table 2.

Table 2. Example of updating distance.

Basic Block	Recalculate or Not	Distance on Old iCFG	Distance on New iCFG
a	Yes	$+\infty$	$2 \cdot \alpha + 1$
b	Yes	$+\infty$	$2 \cdot \alpha$
c	No	$\alpha + 1$	$\alpha + 1$
d	No	α	α

4.5. Illustrating with Examples

In this section, let us discuss the motivating example in Figure 1 again. As we noted above, the edges $d \rightarrow e$ and $d \rightarrow f$ are indirect jumps, and $d \rightarrow e$ determine whether the vulnerability in function *add* can be discovered successfully. If we use LLVM's APIs to construct the CFG for this example, the constructed CFG will not contain the two indirect jump edges, which leads to the failure to discover the vulnerability.

On the contrary, we can find the vulnerability by using our proposed approach. First, we construct the static CFG which is the same as the CFG constructed by existing tools, and still does not contain the indirect jumps. Then, with the indirect jump location (line 16) as the target, a DGF is conducted to attempt to generate test cases that exercise the indirect jump instruction. Through using different test cases to execute and monitor the target program, the indirect jump relations can be resolved. If a test case that satisfies the constraint $a > b$ in line 10 is generated, such as $(a = 10, b = 5)$, the edge $d \rightarrow f$ will be discovered. On the other hand, the edge $d \rightarrow e$ can be discovered when a test case that violates the constraint is generated, such as $(a = 5, b = 10)$. Thus, we can construct a CFG that contains indirect jumps and discover the vulnerability based on the CFG.

For the simple example, it is easy to generate a test case that satisfies the constraint to exercise the indirect jump. In such case, DGF-based and CGF-based test case generations have similar performance. However, in real-world programs, constraints may be so complex that it is difficult to generate test cases that satisfy the constraints through undirected approach. Our DGF-based test case generation can show advantages in handling complex programs, which will be presented in Section 6.

5. Implementation

Based on the proposed approaches, we develop a prototype, dubbed DGF-CFGConstructor. The implementation details of each component are as follows.

Static CFG Construction: We build up the CG and CFG based on LLVM's IR. In the CG, functions are taken as the nodes of the graph and identified by functions' signatures. We discover the CG's edges through analyzing all call instructions. Specifically, if the jump address of a call instruction can be determined by the LLVM's API $CallInst :: getCalledFunction$, the jump relation should be added to the CG as an edge. Otherwise, the call instruction is considered as an indirect jump location. In addition, we can obtain the CFG of each function by another LLVM's API $llvm :: WriteGraph$. Then, the static iCFG is constructed by combining the CG and CFGs according to the method in Section 4.1.

DGF-based Test Case Generation: The DGF-based test case generation is established on AFLGo. The indirect jump locations are considered as the target sites. We calculate the basic block-level distance calculation in a Python script and instrument the seed distance calculation in a LLVM's *pass*. In order to balance the exploration and exploitation, we retain the annealing-based power schedules in AFLGo.

Indirect Jump Instrumentation and Monitoring: The DGF-Constructor records the executed indirect jump relations by instrumenting at the indirect jump locations and the entry of each function. The instrumentation is implemented in another LLVM's *pass*. When a indirect jump is triggered, the relevant jump relation is recorded to a file. After all test cases are executed, we can obtain all indirect jump relations discovered at runtime in the result file.

It should be noted that although the prototype is implemented to analyze the source code at present, in principle, our techniques are also suitable for binary code. We will extend it to binary analysis by replacing the corresponding tools in the further work.

6. Evaluation

In this section, we evaluate our approach on various real-world programs and compare it with related approaches.

6.1. Evaluation Setup

We designed the experiments to answer the following four research questions:

RQ1: Does the DGF-based test cases generation work as expected to generate more test cases that exercise indirect jumps than existing approaches?

RQ2: How effective is the iterative feedback mechanism in resolving indirect jumps?

RQ3: How complete is the CFG constructed by our approach?

RQ4: What are the benefits of applying our approach to program analysis?

Evaluation Benchmarks. We evaluated DGF-CFGConstructor with 8 real-world programs; the relevant information is shown in Table 3. These benchmarks are selected according to the evaluations in the related works [28,29,35,36].

Table 3. The information about the benchmarks.

Project	Version	Description	LoC	Program	parameter
libxml2	2.9.10	Xml parser	496 k	xmllint	–valid –recover file
JasPer	2.0.16	Picture handler	33 k	imgcmp	-f sample -F result
libming	0.4.8	Swf parser	96 k	swftophp	swf_file
FreeType	2.10.0	Library to render fonts	219 k	testsuite	file
libpng	1.6.37	Png handler	97 k	pngtest	-m file
ImageMagick	7.0.8.6	Picture handler	553 k	magick	covert file result
libjpeg-turbo	1.5.1	Image codec	91 k	djpeg	switches file
libtiff	4.1.0	Tiff processor	138 k	tiffsplit	file -o /dev/null

Experimental Infrastructure. All experiments were conducted on a machine equipped with Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz with 8 cores and 64 GB RAM, running 64-bit Ubuntu LTS 16.04.

6.2. Evaluation of DGF in CFG Construction (RQ1)

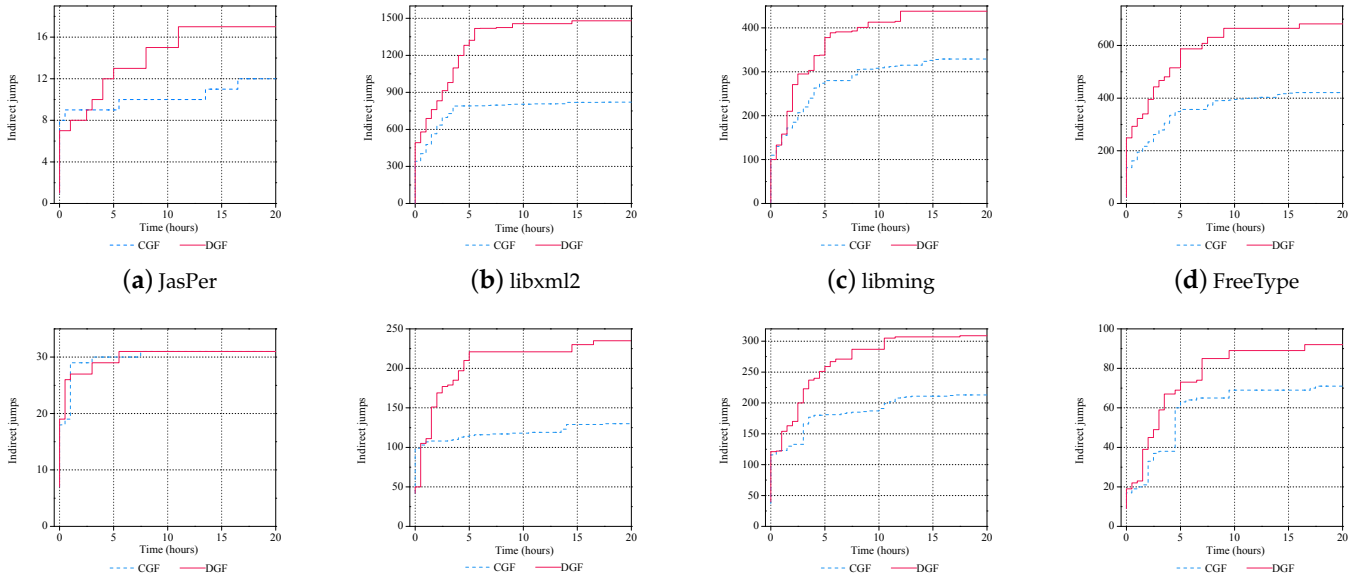
In order to evaluate the effectiveness of DGF in discovering indirect jumps, we implement and compare two CFG constructors.

DGF-CFGConstructor (ours). It is our prototype that is implemented based on our proposed approaches, i.e., the DGF-based test case generation and iterative feedback mechanism. In the DGF-based test case generation, we adopt the annealing-based power schedules and set *time-to-exploitation* to one hour.

CGF-CFGConstructor. It is identical to our prototype except for the test case generation approach. The CFG constructor employs the CFG-based approach proposed in [26].

We use the two CFG constructors to analyze the eight benchmarks for 20 h. Figure 7 shows the trend of the number of indirect jumps discovered by the two approaches in the experiment. Figure 8 shows the statistical results after 20 h. Based on the results, we observe the following cases.

- In the early phase of the analysis (about the first 2 h), the number of indirect jump relations discovered by DGF-CFGConstructor (the red line) is similar to that discovered by CGF-CFGConstructor (the blue line). The reason is that DGF-CFGConstructor adopts the annealing-based power schedules. The weight of the seed distance in energy assignment is zero at the beginning and increases with the testing time. Therefore, the two implementations have similar performance in the early phase.
- After the early phase, the number of indirect jump relations discovered by DGF-CFGConstructor on 7 out of 8 benchmarks (except libpng) shows a better increase than that discovered by CFG-CFGConstructor.
- The statistical results in Figure 8 shows that DGF-Constructor (the red bar) discovers more indirect jump relations than CFG-Constructor (the blue bar). For example, in the analysis of libxml2, CGF-CFGConstructor discovers 821 indirect jump relations. DGF-CFGConstructor is able to discover 1480 indirect jump relations, 80.3% more than CGF-CFGConstructor. On average, DGF-CFGConstructor discovers 61.9% more indirect jump relations than CGF-CFGConstructor on the eight benchmarks.
- In the analysis of libpng, DGF-CFGConstructor and CGF-CFGConstructor discover similar number of indirect jump relations, which is not as expected. The reason may be that there are too few indirect jumps in libpng. The newly discovered indirect jumps cannot provide more feedback to DGF-based test case generation. Therefore, the two approaches show similar performance on this benchmark.



(**a**) JasPer (**b**) libxml2 (**c**) libming (**d**) FreeType

(**e**) libpng (**f**) ImageMagick (**g**) libjpeg (**h**) libtiff

Figure 7. The number of indirect jumps resolved by two approaches on 8 benchmarks. The red line represents our DGF-based approach (DGF-CFGConstructor), and the blue line represents the CGF-based approach (CGF-CFGConstructor).

The main reason for the poor performance of CGF-CFGConstructor in discovering indirect jumps is that CGF aims to generate test cases covering the whole program. The approach assumes that all paths have the same priority and performs a undirected path exploration. This wastes a lot of time to test the code unrelated to indirect jumps. We introduce DGF-based test cases generation to mitigate this problem. The result shows that our approach discovers more indirect jump relations than the existing approach on most benchmarks.

Overall, the analysis indicates that the answer to RQ1 is definite: the DGF-based approach is able to resolve more indirect jump relations than the existing approach.

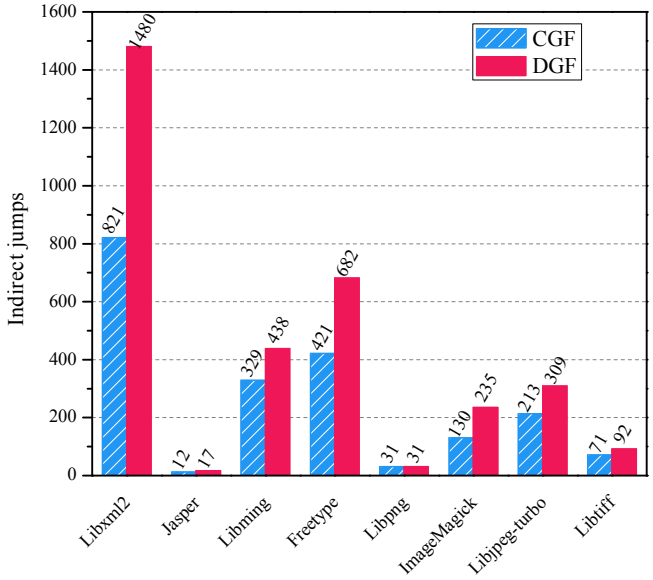


Figure 8. The total number of indirect jumps resolved by two approaches on 8 benchmarks after 20 h. The red bar represents our DGF-based approach (DGF-CFGConstructor), and the blue bar represents the CGF-based approach (CGF-CFGConstructor).

6.3. Evaluation of the Iterative Feedback Mechanism (RQ2)

In this section, we make an investigation about the effectiveness of our proposed iterative feedback mechanism. Another CFG constructor is implemented as a comparison.

NFB-CFGConstructor. It is identical to our prototype DGF-CFGConstructor, except that the iterative feedback mechanism is removed.

In the experiment, two appraisal indexes are used to evaluate the iterative feedback mechanism. The number of indirect jumps discovered is still one of them. In order to quantify the sustainable performance of the approaches, we present *stuck time* as another appraisal index. "Stuck" indicates a state in the process of program testing, in which it is difficult to discover new paths. This concept is mentioned in previous works [7,37], but not defined accurately. We attempt to give a quantitative definition about "stuck" state in discovering indirect jumps.

$I(t_0, t_1)$ is used to represent the number of indirect jump relations discovered from time t_0 to t_1. If there is a time t_s, for any time t $(t \geq t_s)$, the condition

$$I(t, t + \Delta t) / I(0, t) < p \tag{4}$$

is always satisfied, then the testing is considered to get "stuck" at t_s. In the condition, Δt and p are given constants, representing the time interval and the threshold, respectively. *Stuck time* is defined as the minimum of the time satisfying the above condition and denoted as T_s. Intuitively, *stuck time* represents the start time after which the test can hardly discover indirect jump relations.

Figure 9 shows the analysis results of CGF-CFGConstructor and NFB-CFGConstructor on the eight benchmarks. In the experiment, Δt is set five hours and the threshold p is set 5%.

In the early phase of the test, DGF-CFGConstructor (the red line) and NFB-CFGConstructor (the green line) have the similar performance. After the early phase, DGF-CFGConstructor has a more obvious increase than NFB-CFGConstructor. On the all benchmarks, NFB-CFGConstructor gets stuck earlier than CGF-CFGConstructor. When the approach without iterative feedback mechanism gets stuck, our approach can still resolve new indirect jumps continuously. This is because that the iterative feedback updates the seed distance with new structure information and allows our approach to continuously generate test cases that exercise indirect jumps.

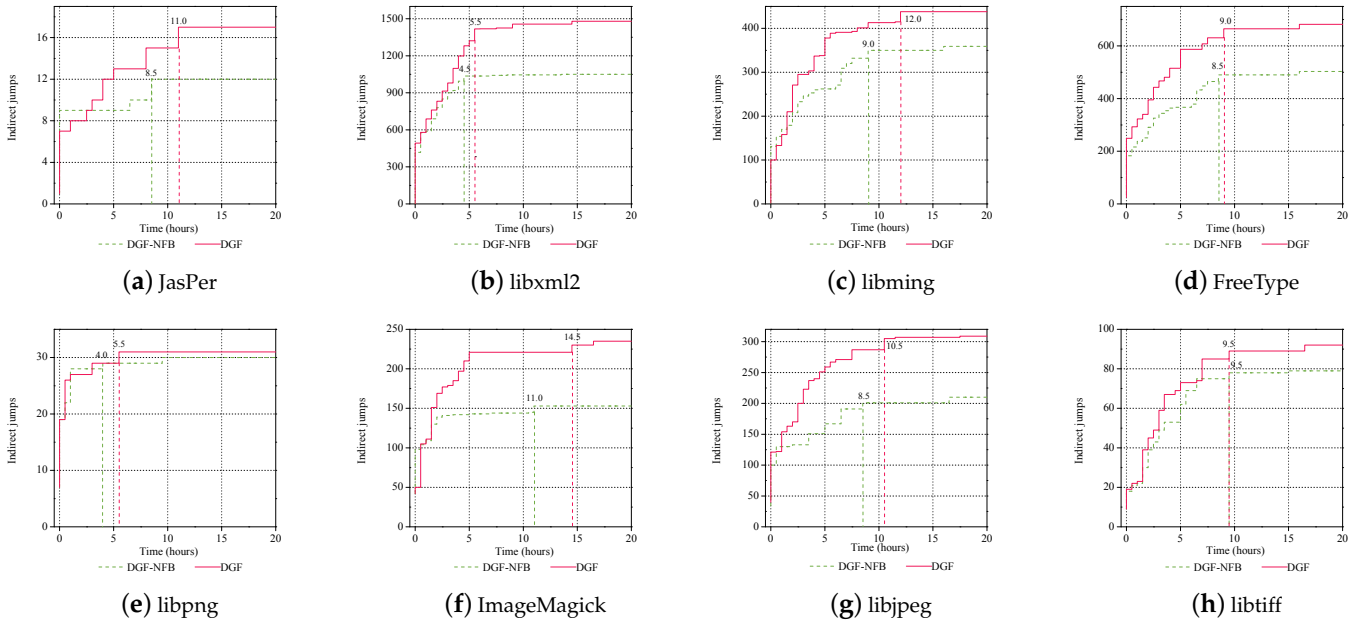


Figure 9. The number of indirect jumps resolved by two approaches on 8 benchmarks. The red bar (DGF) represents our approach with iterative feedback, and the green bar (DGF-NFB) is the opposite. The vertical line represents the *stuck time* of the corresponding approach.

Figure 10 shows the statistical results of the two approaches after 20 h. On the all eight benchmarks, DGF-CFGConstructor (the red bar) can resolve more indirect jumps than NFB-CFGConstructor (the green bar). More specifically, our approach discovers 37.1% more indirect jump relations than the approach without the iterative feedback mechanism.

Overall, the analysis indicates that the answer to RQ2 is definite; our iterative feedback mechanism can improve the sustainability of test case generation and the ability of resolving indirect jumps.

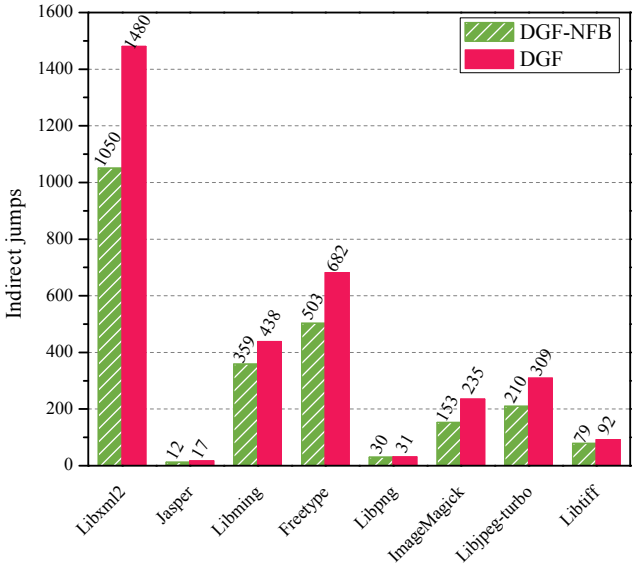


Figure 10. The total number of indirect jumps resolved by two approaches on 8 benchmarks after 20 h. The red bar (DGF) represents our approach with iterative feedback, and the green bar (DGF-NFB) is the opposite.

6.4. Completeness of the CFGs (RQ3)

In the above experiments, we evaluate the effectiveness of two key techniques in our approach. In this section, we further evaluate our prototype against existing approaches and attempt to answer the research question: how complete is the CFG constructed by our approach?

We compare DGF-Constructor with the following tools:

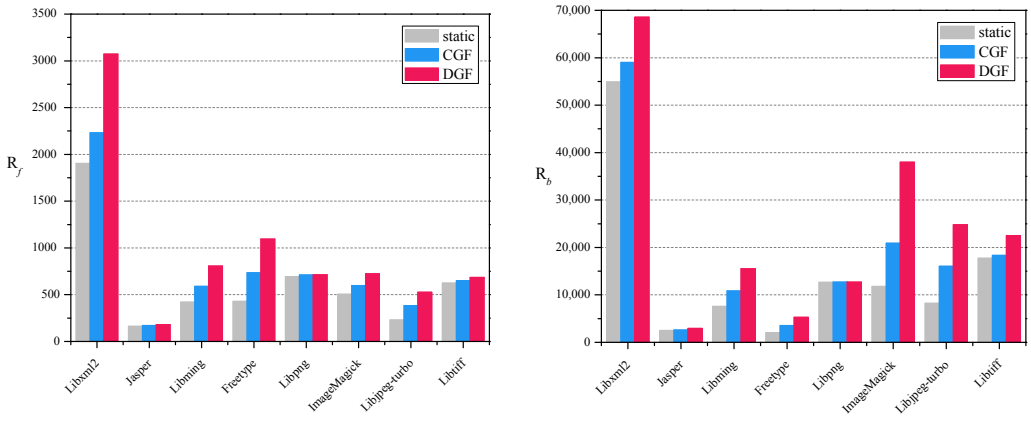
Static-CFGConstructor. It is a pure static tool implemented according to LLVM's builtin APIs.

CGF-CFGConstructor. It is a hybrid tool based on the approach proposed in [26]. The tool has been used in evaluating the test case generation in Section 6.2.

There is not appraisal index to evaluate the completeness of CFGs in the previous work. In many practical program analyses, such as taint analysis and symbolic execution, analysts usually pay more attention to the code that is reachable from the entry of program on the CFG. Therefore, we adopt *reachable code scale* as an important factor to measure the completeness of CFGs.

More specifically, we present two appraisal indexes (R_f, R_b) to quantify the scale of the reachable code. The first index R_f is the number of functions that can be reachable from the program entry on the iCFG. The second index R_b is the number of reachable basic blocks. On the same eight benchmarks, we use three approaches to conduct the experiment: the static analysis (Static-CFGConstructor), the CGF-based approach (CGF-CFGConstructor), and our DGF-based approach (DGF-CFGConstructor). The experiment lasted for 20 h.

Figure 11 shows the reachable code scale of the CFGs constructed by the three approaches. On most benchmarks, DGF-CFGConstructor can discover more reachable functions and basic blocks than the other two tools. Precisely, our approach discovers 62.9% more reachable functions and 94% more reachable basic blocks than Static-CFGConstructor. Compared with CGF-CFGConstructor, our approach discovers 23.9% more reachable functions and 34.9% more reachable basic blocks.



(**a**) The number of reachable functions. (**b**) The number of reachable basic blocks.

Figure 11. The reachable code scale of CFGs constructed by static-based, CFG-based, and DGF-based approaches.

Overall, the analysis indicates that the answer to RQ3 is definite. Our approach is able to construct more complete CFGs than other approaches.

6.5. Application: Security Analysis (RQ4)

In this paper, we focus on how to construct more complete CFGs, which is a prerequisite for many program analyses. There are many applications that may benefit from our work, such as program verification, vulnerability detection, and code similarity analysis. At present, except vulnerability detection, we have not sufficiently evaluated our approach in specific applications, as it will take a lot of effort and knowledge to evaluate various

applications. Therefore, in this section, we take the vulnerability detection as an example to illustrate the practical significance of our work.

Figure 12 shows a real vulnerability (CVE-2015-5221) [38] hidden in the image processing library JasPer. The vulnerability is located deep in the program and may be triggered only when the specific call sequence is executed, i.e., $main \rightarrow jas\_image\_decode \rightarrow mif\_decode \rightarrow mif\_hdr\_get \rightarrow mif\_process\_cmpt$.

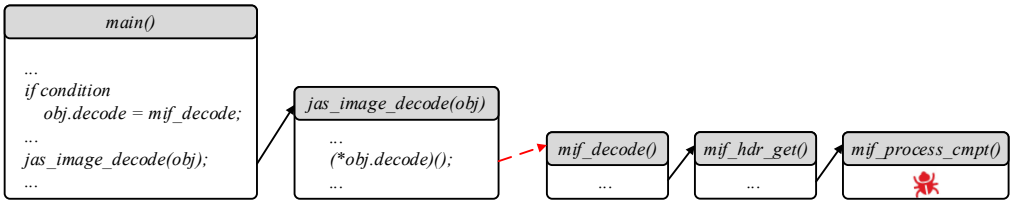


Figure 12. Example of updating the distance incrementally.

Unfortunately, in the call sequence $jas\_image\_decode \rightarrow mif\_decode$ is an indirect jumps. If analysts conduct the security analysis based on the CFGs constructed by the traditional static tool, they may not be able to resolve these indirect jumps and miss the vulnerability. On the contrary, our approach can construct more complete CFGs containing this indirect jump relation and help analysts discover this vulnerability.

Although DGF-CFGConstructor is still a prototype, it has been able to analyze the real-world programs and been applied to vulnerability detection. In the future, we will improve the prototype and make it suitable for more usage scenarios.

6.6. Discussion

In this section, we will discuss the limitations of DGF-CFGConstructor and future directions of research to further improve the completeness of CFGs.

Although our DGF-based approach resolves more indirect jumps than traditional approaches, it still cannot guarantee that all indirect jumps can be exercised. Many factors affect the coverage of DGF, such as the initial seeds and the complexity of the target programs. In future work, we can try to improve the coverage of indirect jumps by improving the seed selection and further analyzing the structure of the target programs. In addition, our approach might benefit from other improvements of DGF.

The scalability of our approach depends on the two techniques, i.e., static CFG construction and dynamic directed gray-box fuzzing (DGF). The two techniques have been proved to be scalable in analyzing large-scale programs in previous researches. Therefore, our approach is also scalable. Specifically, in the static part, LLVM can construct the static CFG for any real-world program in the benchmark in a few minutes, which is trivial compared to dynamic part. On the dynamic part, directed gray-box fuzzing continues to generate test cases until it is aborted manually. Like existing fuzzing works, we cannot give an exact end time. However, intuitively, our algorithm can be aborted when the DGF-based test case generation gets *stuck*, i.e., the *stuck time* is reached. In our evaluation on eight real-world programs, the average stuck time' is about 9.7 h.

At present, DGF-CFGConstructor is still a prototype. It is implemented to analyze source code. However, our proposed DGF-based test cases generation and iterative feedback mechanism can also be applied for binary code. We are extending DGF-CFGConstructor to the binaries.

In addition, the CFG is applied in many practical analyses. However, in this paper, we only preliminarily evaluate the application of our approach in vulnerability detection. In the future, we will extend the approach to other applications, such as malware analysis, code similarity analysis, and so on.

7. Related Works

We summarizes the related works from two aspects: CFG construction and gray-box fuzzing.

7.1. CFG Construction

Static analysis is the most popular approach to construct CFGs in practice. There are many tools based on static analysis, such as IDA pro [20] for binary code and LLVM [22] for source code. They construct CFGs by statically dividing the code into basic blocks and connecting the basic blocks with jump relations. The limitation of this technique is that it is difficult to resolve indirect jump targets by pure static analysis. To handle indirect jumps, Chen et al. [29] proposed to apply the inclusion-based pointer analysis against the function pointers. The point-to set is calculated by focusing on four rules: address-of, copy, assign, and dereference. Due to the inherent difficulty of static analysis to resolve indirect jumps, these approaches cannot produce precise control flow information. In contrast, our approach resolves indirect jumps by dynamic execution and does not suffer from this limitation.

Dynamic analysis is proposed to construct precise CFGs. Traditional dynamic analysis techniques only cover a small portion of program execution paths. Fex [23] uses *forced execution* to improve the coverage of conditional branches. It dynamically tracks conditional branches and save the program state. Then, restore the saved program state and force the execution the other path by manipulating the CPU instruction pointer. Although, this approach effectively improves the branch coverage, the state saving and restoring cause a lot of overhead. Therefore, this approach cannot be applied in large-scale real-world programs efficiently. In contrast, our approach depends on the static analysis and DGF techniques, which are proved to be scalable in real-world programs.

In recent years, hybrid techniques have been proposed to construct CFGs. Babic et al. [24] dynamically compute the indirect jump relations with a set of seed tests and augment the computed relations with statically computed direct jumps. However, they do not pay enough attention to how to generate test cases, which determines the completeness of CFGs. The test cases in their work are generated randomly. Zhu et al. [26] proposed using *coverage-based gray-box fuzzing (CGF)* to generate test cases to handle indirect jumps. However, the approach still cannot construct complete CFGs, because it adopts a undirected path exploration and spends a lot of resources to test the code unrelated to indirect jumps. In contrast, our approach uses DGF instead of CGF to generate test cases and spends most of the time budget on exercising indirect jumps. Therefore, our approach can resolve more indirect jumps and improve the completeness of CFGs.

7.2. Gray-Box Fuzzing

According to the purposes, gray-box fuzzing is divided into coverage-based gray-box fuzzing (CGF) and directed gray-box fuzzing (DGF).

Coverage-based gray-box fuzzing seeks to generate test cases that can achieve the maximum code coverage. AFL [39] is the most popular coverage-based gray-box fuzzer. It adopts a undirected path exploration to cover the code. Many improved techniques [35,36,40,41] are proposed based on AFL. Vuzzer [35] assigns different weights to the execution paths and gives priority to the long paths. AFLFast [40] pays more attention to the low-frequency paths in the fuzzing campaign. CGF is widely applied in vulnerability detection. However, due to the undirected path exploration, this technique wastes a lot of resources on unrelated code.

Directed gray-box fuzzing, proposed by Böhme et al. [28], seeks to spend most of its time on reaching specific target locations without wasting resources on unrelated code. This technique performs a directed path exploration by giving priority to the paths close to the targets. At present, DGF is also mainly applied in vulnerability detection. Existing works [28–34] usually take the changed statements or dangerous locations as target locations. Their purposes are to generate test cases that can trigger vulnerabilities.

However, in this paper, DGF is applied to construct CFGs. We take indirect jump locations as the target locations of DGF and apply this technique to improve the completeness of CFGs.

8. Conclusions

Control flow graph (CFG) is the basic of program analysis, such as malware analysis, vulnerability detection, code similarity analysis, etc. How to resolve indirect jumps is the main challenge of constructing complete CFGs. In this paper, we propose an approach that applies static analysis and dynamic analysis to construct more complete CFGs. We first employ a static analysis to construct the static CFGs without indirect jump relations. Then, in order to resolve indirect jump relations, we propose adopting directed gray-box fuzzing (DGF) instead of coverage-based gray-box fuzzing to generate test cases that exercise indirect jumps. Finally, we combine the static CFGs and indirect jump relations to construct more complete CFGs. In addition, we also propose an iterative feedback mechanism to further improve the completeness of CFGs.

We implement a prototype, named DGF-CFGConstructor, for constructing CFGs and evaluate it on real-world programs. The experimental results have shown that DGF-CFGConstructor can resolve more indirect jumps and construct more complete CFGs than the state-of-the-art CFG constructors. We believe that it provides a promising foundation for precise analysis of programs.

Author Contributions: Conceptualization, Y.L. and K.Z.; methodology, K.Z., H.H., and L.Y.; software, L.Y. and J.Z.; validation, K.Z. and H.H.; investigation, K.Z.; resources, Y.L.; writing—original draft preparation, K.Z.; writing—review and editing, Y.L., J.Z., and K.Z.; supervision, Y.L.; project administration, Y.L. All authors read and agreed to the published version of the manuscript.

Funding: This research was supported by National Key Research and Development Project of China (No. 2017YFB0802900).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: We would like to sincerely thank the reviewers for your insightful comments that help us improve this work.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Allen, F.E.; Cocke, J. A program data flow analysis procedure. *Commun. ACM* **1976**, *19*, 137. [CrossRef]
2. Späth, J.; Ali, K.; Bodden, E. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* **2019**, *48*, 29. [CrossRef]
3. Grech, N.; Smaragdakis, Y. P/Taint: Unified points-to and taint analysis. *Proc. ACM Program. Lang.* **2017**, *102*, 28. [CrossRef]
4. Wang, T.; Wei, T.; Gu, G.; Zou, W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP), Oakland, CA, USA, 16–19 May 2010; pp. 497–512.
5. King, J.C. Symbolic execution and program testing. *Commun. ACM* **1976**, *19*, 385–394. [CrossRef]
6. Baldoni, R.; Coppa, E.; D'elia, D.C.; Demetrescu, D.; Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv.* **2018**, *50*, 39. [CrossRef]
7. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016.
8. Fetzer, J.H. Program verification: The very idea. *Commun. ACM* **1988**, *31*, 1048–1063. [CrossRef]
9. Si, X.; Dai, H.; Raghothaman, M.; Song, L. Learning loop invariants for program verification. *Adv. Neural. Inf. Process. Syst.* **2018**, *31*, 7751–7762.
10. Bruschi, D.; Martignoni, L.; Monga, M. Detecting self-mutating malware using control-flow graph matching. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Berlin, Germany, 13–14 July 2006; pp. 129–143. [CrossRef]
11. Alam, S.; Traore, I.; Sogukpinar, I. Annotated control flow graph for metamorphic malware detection. *Comput. J.* **2015**, *10*, 2608–2621. [CrossRef]

12. Ma, Z.; Ge, H.; Liu, Y.; Zhao, M.; Ma, J. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE Access* **2019**, *7*, 21235–21245. [CrossRef]
13. Iadarola, G.; Martinelli, F.; Mercaldo, F.; Santone, A. Call graph and model checking for fine-grained android malicious behaviour detection. *Appl. Sci.* **2020**, *10*, 7975. [CrossRef]
14. Sun, X.; Zhongyang, Y.; Xin, Z.; Mao, B.; Xie, L. Detecting code reuse in android applications using component-based control flow graph. In Proceedings of the International Conference on Information Security and Privacy Protection, Marrakech, Morocco, 25–27 May 2014; pp. 142–155. [CrossRef]
15. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D. Neural network-based graph embedding for cross-platform binary code similarity detection. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), Dallas, TX, USA, 30 October–3 November 2017; pp. 363–376. [CrossRef]
16. Novak, M.; Joy, M.; Kermek, D. Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Trans. Comput. Educ.* **2019**, *27*, 1–37. [CrossRef]
17. Jana, S.; Kang, Y.; Roth, S.; Ray, B. Automatically detecting error handling bugs using error specifications. In Proceedings of the 25th USENIX Security Symposium, Austin, TX, USA, 10–12 August 2016; pp. 345–362.
18. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
19. Chang, P.; Hao, E.; Patt, Y.N. Target prediction for indirect jumps. In Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97), Denver, CO, USA, 1 June 1997; pp. 274–283.
20. Hex-Rays. IDAPro Disassembler. Available online: https://www.hex-rays.com/ (accessed on 20 December 2020).
21. Bardin, S.; Herrmann, P.; Leroux, J.; Ly, O.; Tabary, R.; Vincent, A. The BINCOA framework for binary code analysis. In Proceedings of the 23rd International Conference (CAV 2011), Snowbird, UT, USA, 14–20 July 2011.
22. The LLVM Compiler Infrastructure. Available online: https://llvm.org/ (accessed on 20 December 2020).
23. Xu, L.; Sun, F.; Su, Z. *Constructing Precise Control Flow Graphs from Binaries*; Technical Report; Technical Report CSE-2009-27; University of California: Davis, CA, USA, 2009.
24. Babić, D.; Martignoni, L.; McCamant, S.; Song, D. Statically-directed dynamic automated test generation. In Proceedings of the 2011 International Symposium on Software Testing and Analysis, Toronto, ON, Canada, 17–21 July 2011; pp. 12–22. [CrossRef]
25. Nguyen, M.H.; Nguyen, T.B.; Quan, T.T.; Ogawa, M. A hybrid approach for control flow graph construction from binary code. In Proceedings of the 20th Asia-Pacific Software Engineering Conference, Bangkok, Thailand, 2–5 December 2013; pp. 159–164.
26. Zhu, K.; Lu, Y.; Huang, H.; Deng, Z.; Deng, Y. Construction approach for control flow graph from binaries using hybrid analysis. *J. Zhejiang Univ.* **2019**, *53*, 829–836.
27. Ye, Z.; Jiang, X.; Shi, D. Combined method of constructing binary-oriented control flow graphs. *Appl. Res. Comput.* **2018**, *35*, 2168–2171.
28. Böhme, M.; Pham, V.; Nguyen, M.; Roychoudhury, A. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), Dallas, TX, USA, 30 October–3 November 2017; pp. 2329–2344. [CrossRef]
29. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), Toronto, ON, Canada, 15–19 October 2018; pp. 2095–2108. [CrossRef]
30. Wang, H.; Xie, X.; Li, Y.; Wen, C.; Li, Y.; Liu, Y.; Qin, S.; Chen, H.; Sui, Y. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20), Seoul, Korea, 24 June–16 July 2020; pp. 999–1010. [CrossRef]
31. Wen, C.; Wang, H.; Li, Y.; Qin, S.; Liu, Y.; Xu, Z.; Chen, H.; Xie, X.; Pu, G.; Liu, T. MemLock: Memory usage guided fuzzing. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20), Seoul, Korea, 24 June–16 July 2020; pp. 765–777. [CrossRef]
32. Nguyen, M.D.; Bardin, S.; Bonichon, R.; Groz, R.; Lemerre, M. Binary-level directed fuzzing for use-after-free vulnerabilities. In Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20), San Sebastian, Spain, 14–16 October 2020.
33. Ye, J.; Li, R.; Zhang, B. RDFuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation. *Math. Probl. Eng.* **2020**, *2020*, 7698916. [CrossRef]
34. Aschermann, C.; Schumilo, S.; Abbasi, A.; Holz, T. Ijon: Exploring deep state spaces via fuzzing. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–20 May 2020; pp. 1597–1612.
35. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware evolutionary fuzzing. In Proceedings of the 2017 NDSS Symposium, San Diego, CA, USA, 26 February–1 March 2017.
36. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In Proceedings of the IEEE Symposium on Security and Privacy(SP), San Francisco, CA, USA, 20–24 May 2018.
37. Zhao, L.; Duan, Y.; Yin, H.; Xuan, J. Send hardest problems my Way: Probabilistic path prioritization for hybrid fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 24–27 February 2019.
38. CVE-2015-5221: An Use-after-Free Vulnerability in the JasPer JPEG-2000 Library before 1.900.2. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5221 (accessed on 20 December 2020).

39. Michal, Z. American Fuzzy Lop (AFL). Available online: http://lcamtuf.coredump.cx/afl/ (accessed on 20 December 2020).
40. Böhme, M.; Pham, V.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* **2017**, *45*, 489–506. [CrossRef]
41. Wang, M.; Liang, J.; Chen, Y.; Jiang, Y.; Jiao, X.; Liu, H.; Zhao, X.; Sun, J. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18), Gothenburg, Sweden, 27 May–3 June 2018; pp. 61–64. [CrossRef]