MDPI

Article

ObFuzzer: Object-Oriented Hybrid Fuzzer for Binaries

Xinglu He , Pengfei Wang *, Kai Lu and Xu Zhou

College of Computer, National University of Defense Technology, Changsha 410073, China
* Correspondence: pfwang@nudt.edu.cn

Abstract: In recent years, coverage-guided technology has become the mainstream method of fuzzing. A coverage-guided fuzzer can guide a program to a new path (edge) so that previously untested code can be tested. As coverage-guided fuzzers have become more popular, the difficulty of discovering vulnerabilities has increased significantly. This paper proposes ObFuzzer, an object-oriented binary hybrid fuzzer based on a new assumption. Namely, the object which has been operated more times and operated in more positions is more likely to have defects. Our ObFuzzer consists of the following steps. First, ObFuzzer obtains the inner relations of object operations in the target program through static analysis and analyzes the riskiness of the basic blocks containing such operations. Then, ObFuzzer generates test cases that can guide the program to the basic blocks that this paper considers to be the most dangerous by symbolic execution. Finally, fuzzing is performed using the riskiness of the object operations rather than code coverage. To demonstrate the effectiveness of ObFuzzer over a traditional coverage-guided fuzzer, this paper evaluates its performance in a real program. When facing object-oriented programs, ObFuzzer has a 29% to 40% increase in object operation complexity during execution. These more complex object operations can enhance the ability to discover vulnerabilities related to object operations. Eventually, ObFuzzer found five unique vulnerabilities and one logic error without a crash in "xpdf".

Keywords: vulnerability discovery; hybrid fuzzer; static analysis; symbolic execution; object-oriented fuzzing

Citation: He, X.; Wang, P.; Lu, K.; Zhou, X. ObFuzzer: Object-Oriented Hybrid Fuzzer for Binaries. *Appl. Sci.* **2022**, *12*, 9782. https://doi.org/10.3390/app12199782

Academic Editor: Luis Javier Garcia Villalba

Received: 22 July 2022
Accepted: 19 September 2022
Published: 28 September 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Software vulnerabilities have become more and more critical to our daily lives. To solve this problem, fuzzing has become the most popular technology for vulnerability discovery. The traditional fuzzing technology randomly generates the test cases and detects the target program status. It can nearly be used in all programs. However, with pure randomly generated test cases, it is hard to test all code of the target program. Many test cases lead the target program to the same path (exception handling, etc.) and other paths are hard to test.

Due to this drawback of purely random fuzzing, grey box fuzzing has been proposed [1]. When the target program is being executed, the fuzzer can obtain information from it. With this information, the fuzzer can generate more effective test cases and increase the vulnerability-discovery performance. Coverage-guided has been the mainstream approach to grey box fuzzing. Its basic assumption is that unexecuted code is more likely to have defects. Coverage-guided can help the fuzzer increase the coverage of basic blocks or edges. It assists the vulnerability-discovery tool in testing untested positions in the target program. This technique has found a large number of vulnerabilities in recent years. However, software security and vulnerability discovery capability are improving each other. Many software developers perform coverage-based fuzzing on their software before it is released. This leads to making it increasingly difficult for the coverage-based fuzzer to find vulnerabilities in programs.

The traditional coverage-guided fuzzing technology only uses the edge-based coverage measure method. Therefore, it only effectively discovers the vulnerabilities generated

in the same base block or adjacent basic blocks. This paper summarized three types of vulnerabilities that are difficult to find by coverage-based vulnerability-discovery tools.

1. Overflow: Vulnerability is related to the loop: Since the loop has a low impact on coverage, such vulnerabilities are difficult to discover;
2. Null Pointer Dereference: Vulnerability is related to a series of discontinuous basic block execution orders: Since no edge can reflect the execution order of the disconnected basic blocks, such vulnerabilities are difficult to discover;
3. Type Conversion: Vulnerability is related to the current state of the object: Since the state of the object cannot be reflected by coverage, such vulnerabilities are difficult to discover.

It is difficult to find an edge that is needed in traditional coverage-guided fuzzing to guide effective discovery of these vulnerabilities. Therefore, a completely new technique is needed to improve the effectiveness of grey box vulnerability discovery.

This paper proposes a new vulnerability-discovery technology called **object-oriented vulnerability discovery** to address these challenges. The object-oriented vulnerability-discovery technology allocates its resource in accordance with the object operation information obtained during the static analysis phase. This paper applies this standard in designing ObFuzzer: a binary hybrid fuzzer that combines static analysis, symbolic execution and fuzzing. ObFuzzer can discover the vulnerabilities which traditional coverage-guided fuzzers find it difficult to discover. First, ObFuzzer's static analysis phase obtains the nodes (basic blocks) related to object operations in the program by static analysis and scores the corresponding nodes. It will score these nodes with three types of scoring: PCP (Potential Crash Point), PCPP (Potential Crash Point' Predecessor) and PEP (Potential Error Point) scores. The scoring approach will be explained later. Second, ObFuzzer's symbolic execution phase generates test cases that can be executed to the position containing complex object operations by symbolic execution. This allows bypassing a large number of positions that are not related to object operations and testing the program in a more targeted way. Finally, ObFuzzer's fuzzing phase generates test cases with more complex object operations through object operation complexity guide methods. Then, it tries to discover vulnerabilities in the program caused by complex object operations.

In summary, this paper contributes the following:

First, this paper is inspired by the idea of procedure orientation and object orientation. The traditional coverage-based vulnerability-discovery technique is classified as procedure-oriented vulnerability discovery and the object-oriented vulnerability-discovery approach is creatively proposed.

Second, this paper aims to achieve object-oriented vulnerability discovery. It proposes a new scoring approach for object operations in the program by static analysis. By scoring in this way, it is possible to show more visually the inner relations between object operations in the program.

Third, this paper aims to be able to guide the program execution to the high-risk locations efficiently. This paper innovatively proposes ExeTree, which assists ObFuzzer in path planning.

Fourth, this paper further converts the traditional procedure-oriented vulnerability-discovery approach based on coverage guidance into a new object-oriented vulnerability-discovery approach based on object operation complexity guidance.

Finally, we implemented the prototype tool ObFuzzer for the proposed object-oriented approach and evaluated it with a real-world program. Experimental results show that ObFuzzer has a 29% to 40% increase in object operation complexity when facing object-oriented programs.

Eventually, ObFuzzer found five unique vulnerabilities and one logic error without a crash in "xpdf".

2. Background

Fuzzing has become a mainstream approach in recent years due to the presence of AFL. AFL has set the standards for the fuzzing field with its approach to program execution, information acquisition and coverage comparisons. Many technologies have been proposed to increase the ability of coverage-guided vulnerability discovery. CollAFL solves the coverage collide problem [2]. With more precise coverage, the coverage-guided fuzzer can obtain better results. INSTRIM [3], INSTRCR [4] and ZAFL [5] use highly efficient compilation techniques to improves target program execution speed. The "laf-intel" changes the binary roadblocks and tests the code behind roadblocks [6]. Angora uses a principled search to solve the magic bytes in programs [7]. VUZZER uses static analysis and taints tracking methods to improve the test cases' efficiency of coverage-guided vulnerability discovery [8]. PIN [9] and QEMU [10] use the dynamic instrumentation technology to extend the coverage-guided fuzzer to closed-source targets. PTFuzzer uses Intel Processor Trace technology to obtain trace information with low overhead on closed-source targets [11]. NEUFuzz uses deep learning technology reordering seeds to increase the finding vulnerability possibilities [12]. ParmeSan [13] and AddressSanitizer [14] use Sanitizer as a new guidance method to try to trigger potential vulnerabilities in the program. Avatar [15], Avatar2 [16] and IoTFuzzer [17] extend traditional vulnerability discovery to embedded devices. P-Fuzz uses parallelization to improve performance utilization in fuzzing environments more effectively, enhancing the fuzzer performance [18].

As fuzzing techniques become more developed, their ability to discover vulnerabilities decreases. Its original high-performance advantage is no longer enough to compensate for the difficulty of discovering new deficiencies in pure coverage guidance at this stage. Therefore, to overcome the shortcoming in traditional coverage-based fuzzing, hybrid fuzzing is proposed. Static analysis and symbolic execution are added to traditional fuzzing to assist fuzzing in discovering new vulnerabilities. Driller [19] provides a simple hybrid fuzzing idea. It introduces symbolic execution to assist in improving the efficiency of vulnerability discovery when it is difficult to improve coverage with fuzzing. QSYM [20] focuses on making symbolic execution more suitable for hybrid fuzzing usage cases. PANGOLIN's [21] goal is to improve the reuse rate of information obtained during the symbolic execution phase and to use the results of constraint solving to guide test case generation in the fuzzing phase. SAVIOR [22] proposes a bug-driven hybrid fuzzing approach. It focuses more attention on the parts of the code that have bugs by the Undefined Behavior Sanitizer. HFL [23] improves the applicability of hybrid fuzzing and introduces hybrid fuzzing into kernel vulnerability discovery. How to combine multiple techniques more effectively to discover more vulnerabilities has become a hot research point in this field.

3. Motivation

Coverage-guided technology has advantages in vulnerability discovery. Therefore, this technology is widely used in the field of software security. As a result, vulnerabilities easily discovered by this technology have been found in the past. Discovering vulnerabilities with this technology is becoming increasingly difficult. However, vulnerabilities that are not easily discovered with this technology also exist in programs. This paper uses the following three examples to briefly describe some common vulnerabilities not easily discovered with coverage-guided technology. Later sections will explain how to solve these types of vulnerabilities with ObFuzzer.

3.1. Overflow

Overflow vulnerability involves out-of-memory reading/writing due to abnormal boundary conditions. It is shown in the example Listing 1.

Listing 1. Overflow.

```
1  int main(int argc, char *argv[])
2  {
3  char str[10];
4  unsigned int length = (unsigned int)argv[1];
5  if (length + 1 >= 10) return 0;
6  ...
7  for (int i = 0; i < length; i++){
8  a[i] = "a";
9  }
10 a[length] = NULL;
11 return 0;
12 }
```

The program wants to set "str[10]" with several "a" characters and end with a NULL pointer. "length" is used to store the length of the "a" character. The program needs to leave a place for the null pointer. "length" + 1 needs to be less than 10. However, when the length is 65535, "length" + 1 equals 0 (in 16-bit length), which satisfies the condition of less than 10. An overflow vulnerability is triggered when this code has been executed 65,535 times.

Traditional coverage-guided (based on edge coverage) fuzzing technology makes it difficult to effectively distinguish the coverage difference caused by different execution times (edge, more than 2). Such problems are difficult to detect in more complex cases caused by recursion or other forms of code-repeating execution. This type of vulnerability often occurs when reading or writing values with an array or heap block by a loop.

3.2. Null Pointer Dereference

Null pointer dereference vulnerability is the pointer pointing to an illegal address due to an incorrect operation. It is shown in the example Listing 2.

Listing 2. Null Pointer Dereference.

```
1  #include <malloc.h>
2  int main(int argc, char *argv[])
3  {
4  char *str;
5  a = (char *)malloc(sizeof(char), 10);
6  ...
7  free(a);
8  ...
9  printf("%s", *a);
10 return 0;
11 }
```

The program allocates memory to the pointer "str". Then, the program inadvertently frees this memory space. When the program uses (printf) the pointer "str" again, a use-after-free vulnerability is triggered.

The positions where the error occurred may not be in the same base block, or the two basic blocks do not connect. The coverage has no edge that effectively reflects such a relationship of basic blocks or vulnerabilities. This type of vulnerability often occurs in programs that have a large number of operations on a pointer.

3.3. Type Conversion

Type conversion vulnerability is data of other types used as pointers due to incorrect type conversions. It is shown in the example Listing 3.

Listing 3. Type Conversion.

```
1  struct basic_type{
2  char name[10];
3  int shorthand_name_pos;
4  };
5  struct expert_type{
6  char name[100];
7  char *shorthand_name_address;
8  };
9  int main(int argc, char *argv[])
10 {
11 basic_type my_type;
12 my_type.name = "my_type_1";
13 my_type.shorthand_name_pos = 3;
14 expert_type my_expert = (expert_type)my_type;
15 my_expert.name = "my_expert_type_1";
16 ...
17 printf("%s", my_expert.shorthand_name_address);
18 return 0;
19 }
```

The program declares two data structures, `"basic type"` and `"expert type"`. `"basic type"` has a `"name"` attribute which can store ten characters and a `"shorthand name pos"` attribute which stores the relative position to the shorthand name. `"expert type"` has a `"name"` attribute which can store 100 characters and a `"shorthand name address"` attribute which stores the absolute position of the shorthand name. When the `"name"` attribute of `"basic type"` does not meet the requirements and the `"name"` attribute needs to be extended, the `"basic type"` needs to be converted to `"expert type"` and a new name rewritten into the `"name"` attribute. The `"shorthand name pos"` in the `"basic type"`, which is used to store the relative position, will be converted to the `"shorthand name address"` in the `"expert type"`, which is used to store the absolute position in memory. When the program tries to read (printf) the data at the `"shorthand name address"` of `"expert type"`, a null pointer dereference occurs. Because this vulnerability is affected by the current state of the corresponding object, the state of the object is more complex than the path coverage. This type of vulnerability often occurs in object-oriented programming.

In summary, with traditional coverage-guided fuzzing technology, it is hard to detect such vulnerabilities.

4. Design

In Section 3, this paper lists three examples. We aim to design and implement a vulnerability-discovery tool that can effectively address the shortcomings of traditional coverage-based vulnerability-discovery tools in the face of such object-related (arrays, memory blocks, classes, etc.) vulnerabilities.

Many vulnerabilities are related to objects. With the objects being passed everywhere in the program, errors may occur anywhere. The traditional coverage-guided (edge) vulnerability-discovery method pays more attention to the execution relationship between the two basic blocks. This kind of vulnerability-discovery method should be called **procedure-oriented vulnerability discovery**, and only focuses on the execution process of the basic blocks and does not pay attention to the intrinsic relationship between the object. The traditional vulnerability-discovery method is procedure-oriented and is not suitable for vulnerabilities related to objects. Therefore, this paper proposes a new vulnerability-discovery method called **object-oriented vulnerability discovery** to discover these types of vulnerabilities.

4.1. Object-Oriented Vulnerability Discovery

The major contribution of this paper is to propose an object-oriented vulnerability discovery idea to address the shortcomings in procedure-oriented vulnerability discovery.

Unlike traditional procedure-oriented vulnerability discovery, which pays more attention to the relationship between the basic blocks during program execution, object-oriented vulnerability discovery pays more attention to how an object is operated in the program. ObFuzzer identifies objects by memory management functions in symbolic execution. This object can be an arbitrary size, from 1 bit to a complex class. Even more complex stacks, queues, binary trees and so on are included in the meaning of the objects described in this paper.

In practice, a vulnerability-discovery tool pays more attention to object data, which has more probability of causing a crash. The previous three examples show that the program's out-of-bounds read, write, or execution may cause a program crash. Therefore, a vulnerability-discovery tool needs to focus on the objects which it is interested in that are read, written or executed in the program.

This kind of vulnerability caused by objects often has a characteristic. The position of the error and the position of the crash are not necessarily close to each other; they are even far away. Therefore, error points and crash points are two critical points in object-oriented vulnerability discovery. As described previously, crash point is the position of the out-of-bounds read, write, or execution at which the program eventually crashes. Error points are difficult to identify by static analysis, symbolic execution and fuzzing. Therefore, our vulnerability-discovery tool needs a new object-based guided technology to assist a fuzzer in discovering such object-oriented vulnerabilities.

The main idea of the object-oriented vulnerability-discovery technique is shown in Figure 1. The life cycle of an object that can trigger a crash involves a normal object undergoing a series of object-related operations (an exception operation order) becoming an abnormal object, and eventually a crash. With traditional vulnerability-discovery tools, it is hard to find the precise order of object operations that can cause an object to go from normal to abnormal. Therefore, this paper needs an object-oriented vulnerability-discovery fuzzer. It can generate a randomized order of object-related operations by an object operation complexity guide. These complex object operations can help to discover vulnerabilities related to object operations. Then, through extensive testing, they try to discover crashes in the program triggered by complex object operations.

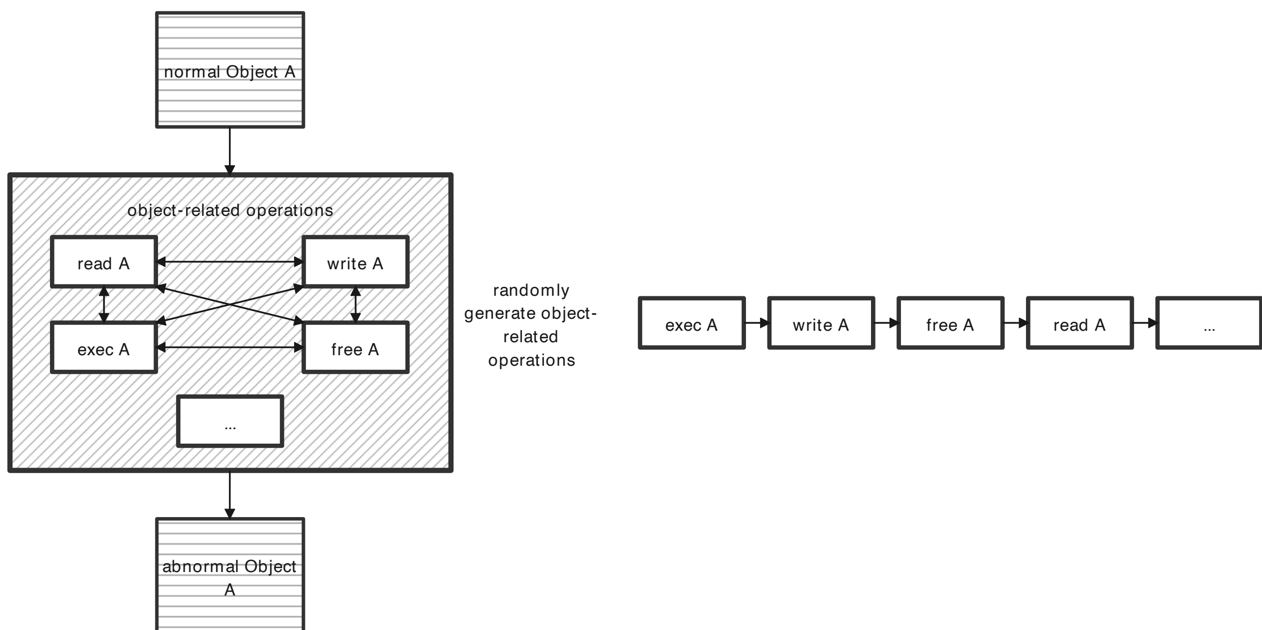


Figure 1. Main Idea.

4.2. Overview

In order to implement an object-oriented approach to vulnerability discovery, we further designed and implemented ObFuzzer.

To achieve object-oriented vulnerability discovery, our vulnerability-discovery tool needs to transform our focus from the relationship between basic blocks in traditional procedure-oriented vulnerability discovery into the relationship between object operations in object-oriented vulnerability discovery.

The difficulties of this transformation include the following three aspects. First, traditional procedure-oriented vulnerability discovery focuses on the relationships between the basic blocks, which can be obtained through simple program analysis or dynamic binary instrumentation. However, the relationships between object operations required in object-oriented vulnerability discovery are often much more difficult to obtain. Second, object-oriented vulnerability discovery begins with a normal object. Many basic blocks which have little relationship with the object have been executed. Therefore, our vulnerability-discovery tool needs to bypass a large number of meaningless basic block tests so that our fuzzer can focus on the basic blocks related to object operations. Third, object-oriented vulnerability discovery should focus on object-related operations. The traditional coverage-guided method is no longer suitable for object-oriented vulnerability discovery and this paper needs to propose a new guide method. It enables an object-oriented fuzzer to more effectively test complex object operations in programs.

This paper proposes a new vulnerability-discovery tool, ObFuzzer (Object-oriented Fuzzer), to achieve object-oriented vulnerability discovery. It is called ObFuzzer and consists of three modules: static analysis, symbolic execution and fuzzing. The overall design of ObFuzzer is shown in Figure 2.

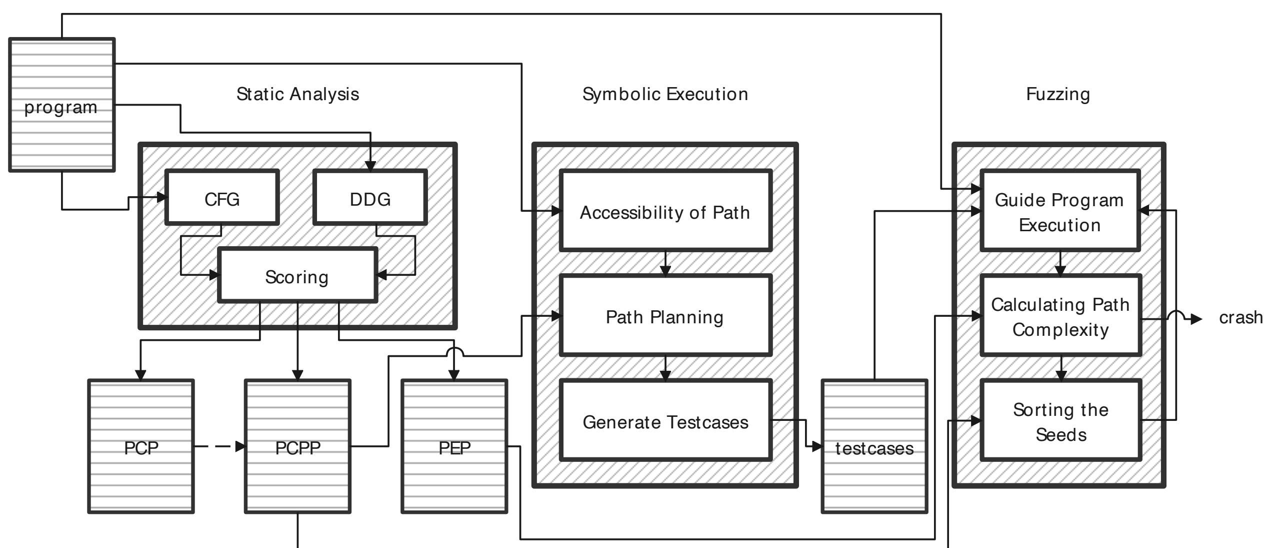


Figure 2. Overview of ObFuzzer.

ObFuzzer uses the static analysis module to analyze the target program and obtain the relationship between the object operations. Through the more detailed static analysis of the program, our understanding of the program goes beyond the traditional relationship between basic blocks to reach the relationship between objects. By effectively using the static analysis results, ObFuzzer can transform from procedure-oriented vulnerability discovery to object-oriented vulnerability discovery. The ObFuzzer's static analysis phase outputs PCP, PCPP and PEP scores.

- The PCP scores are used to calculate the PCPP scores;
- The PCPP scores will be passed to subsequent symbolic execution modules and indicate the basic blocks that the symbolic execution module needs to execute;

- The PEP and PCPP scores will be passed to subsequent fuzzing modules and assist the fuzzing module in implementing the guide based on the object operation complexity.

ObFuzzer uses the symbolic execution module to bypass some complex unwanted basic blocks that have little correlation with object operations. Through the PCPP scores obtained by the static analysis module, the symbolic execution module has learned which basic blocks have more relationships with object operations. Then, ObFuzzer can more effectively bypass a large number of basic blocks ObFuzzer does not care about and quickly guide the program to the basic blocks with high PCPP scores by symbolic execution. Symbolic execution will generate test cases for the target basic blocks and pass them to the fuzzing module as a dictionary. This dictionary helps the fuzzing module bypass a lot of unconcerned basic blocks and focus on object operations.

ObFuzzer uses the fuzzing module to randomly test the object operations of the program. Since the dictionary is generated during symbolic execution, ObFuzzer can quickly reach the basic blocks related to object operations without having to explore paths through a random method, which is slow. ObFuzzer tends to test the basic blocks with more complex object operations by calculating the PEP scores. As a result, the fuzzing module can quickly reach object-related positions and generate more effective test cases that attempt to trigger potential vulnerabilities caused by complex object operations in the program.

The following sections will elaborate on the functions and features of the different modules.

4.3. Static Analysis: Identifying and Scoring Information Related to Object Operations in the Program

In order to implement an object-oriented approach to vulnerability discovery, we propose supporting static analysis and scoring approaches. The basic blocks of the program are scored for their effect in object-oriented vulnerability discovery and prepared for the subsequent dynamic vulnerability discovery.

How to locate, track and analyze objects in the program is a critical issue in object-oriented vulnerability discovery. The program creates and uses objects all the time during execution and it is impractical to analyze every object. Fortunately, ObFuzzer can analyze only the crash-related objects, which significantly narrows the scope of our static analysis.

To define crash-related objects, this paper proposes potential crash points. The potential crash points are the instructions (in practice, we adjust the granularity to the basic block level) that read, write, or execute on memory. As shown in the previous examples, a large number of crashes of object-oriented vulnerabilities occur in incorrect operations on memory. After that, compared to stack memory, heap memory is less frequently and more widely used. So, heap memory's security is more difficult to guarantee. ObFuzzer focuses on operations on heap memory in the static analysis. Therefore, ObFuzzer only recognizes the potential crash points by identifying the read, write and execute operations on the object stored in the heap memory. Since we care about vulnerabilities that crash at the potential crash points, we only need to analyze the data related to the potential crash point pointer. ObFuzzer can prune the basic blocks with little relationship with the potential crash point object and focus only on the basic blocks related to the potential crash point object. In this way, ObFuzzer can shift our focus from the program execution path to the order of the object operations in the program. Thus, the traditional procedure-oriented fuzzing technique can be transformed into the object-oriented fuzzing technique.

The static analysis module mainly completes three functions: identifying the potential crash point of the program, identifying the basic blocks related to the potential crash point's object and scoring previously identified information.

4.3.1. Identifying the Potential Crash Point and the Predecessor of the Potential Crash Point

A static analysis generates a data dependency graph (DDG) and analyzes the instructions for each basic block to identify the instructions that operate on memory. Data in a

heap are usually used across multiple basic blocks or functions; their security is even more difficult to guarantee. Therefore, ObFuzzer identifies data stored in a heap as objects and identifies read, write and execute operations on these objects as potential crash points. To reduce complexity, ObFuzzer prunes the instructions that operate on the stack. Then, ObFuzzer records the position of the memory-related operation and the pointer to the object, which are recognized as potential crash point information.

A potential crash point can be affected by more object operations; the more difficult it is to ensure a potential crash point, the more likely it is a crash will occur.

The control flow graph was constructed and potential crash points were identified. ObFuzzer needs to obtain the basic blocks related to the potential crash point object. Therefore, ObFuzzer obtains the program's control flow graph (CFG). By tracking the data dependencies in the data dependency graph, ObFuzzer obtains information about the basic blocks related to the potential crash point's object.

When a basic block has more data dependencies with more dangerous potential crash points, ObFuzzer tests its subsequent object operations more effectively.

4.3.2. Scoring the Identified Information

Since the ultimate purpose of the static analysis module is to assist the subsequent dynamic analysis tools, it guides the program to the position that is more worthy of testing and tries to test more complex object operations. Therefore, after ObFuzzer identified objects to the potential crash points and the basic blocks related to the potential crash point's object, ObFuzzer needs to score the potential crash points and the basic blocks related to the potential crash point's object.

(1) Scoring the potential crash point

From a programming perspective, if an object is operated by more basic blocks, it is more difficult for the programmer to guarantee its security. Moreover, it is more likely to trigger an object-oriented vulnerability. ObFuzzer needs to conduct more testing on such objects.

We can see from Algorithm 1 that when ObFuzzer scores a potential crash point, ObFuzzer tracks the predecessor node related to the potential crash point on the data dependency graph. It will obtain a higher score when a potential crash point has more predecessor nodes (this can be many layers). Because more predecessor nodes represent that they can be more complex operations, security is more difficult to guarantee.

Algorithm 1: Scoring the potential crash point.

Input: The previously identified potential crash point *PotentialCrashPoint*
Result: The score of potential crash point *PCPScore*

```
1 Worklist ← {PotentialCrashPoint};
2 PCPScore ← 0;
3 for CurrentInfoonDDG in Worklist do
4     Predecessors ← GetPredecessoronDDG(CurrentInfoonDDG);
5     PCPScore ← PCPScore + CountPredecessors(Predecessors);
6     Worklist ← Worklist ∪ Predecessors;
7 end
```

As shown in Figure 3, the number of predecessor nodes of node "A" is 3 and the number of predecessor nodes of node "B" is 1 (data dependency graph). The security of node "A" is more difficult to guarantee than that of node "B". This paper thinks the program is more likely to crash at node "A" than at node "B". Therefore, node "A" should have a higher potential crash point score than node "B". We call this score PCP (potential crash point) score.

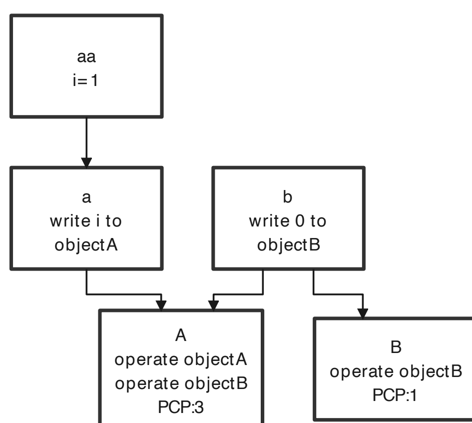


Figure 3. Scoring the potential crash point.

(2) Scoring the potential crash point's predecessor

The premise of causing a program to crash at a potential crash point is that an error occurred in the execution of the program before the potential crash point (in most cases, especially object-oriented vulnerabilities). If the previous program is executed normally, the program will not crash even at the potential crash point. Therefore, the analysis of the predecessor nodes of the potential crash point is significant.

We can see from Algorithm 2 that when ObFuzzer scores a potential crash point's predecessor, ObFuzzer tracks the predecessor nodes of potential crash points on the control flow graph and calculates the PCPP scores for these predecessor nodes. ObFuzzer accumulates the PCPP scores from the successor nodes to the predecessor nodes with the PCP scores uniquely. The unique PCP scores are achieved with the Hash method. Only for a PCP Hash that does not exist in the Hashlist can the score be accumulated. If a basic block can reach more potential crash points and these potential crash points have a higher PCP score, this paper thinks there is a higher probability of composing a valid order of object operations to try to trigger a crash of a potential crash point. Therefore, in calculating the potential crash point's predecessor scores, ObFuzzer starts with the potential crash point and iterates over the predecessor node of the potential crash point. Then it accumulates the successor node's PCP score of the current node.

As shown in Figure 4a, node "a" has only one successor node with a PCP score of 3, node "b" has a successor node with a PCP score of 3 and a successor node with a PCP score of 2 and node "aa" has a successor node with a PCPP score of 3 and a successor node with a PCP score of 1 (control flow graph). Fuzzing from node "b" can reach more potential crash points with a higher PCP score and is more likely to crash than fuzzing from node "a" and node "aa". Therefore, node "b" should have a higher potential crash point's predecessor score than node "a" and node "aa". We call this score the PCPP (potential crash point's predecessor) score.

As shown in Figure 4b, PCPP scores will not be increased if the score from the PCP has been raised before. For example, if the PCPP score of node "a" already includes the PCP score of node "C" when calculating the PCPP score, then the predecessor node "aa" will not recalculate the PCP score of node "C". This ensures that PCPP scores do not accumulate indefinitely. In the loop, when the PCPP score of the predecessor is not less than the successor, further analysis is not required.

Potential crash points have a large number of predecessor nodes. The score of the predecessor node is calculated by accumulating the scores of the successor nodes. Therefore, the score of the predecessor node must not be less than that of the successor node. ObFuzzer prefers to use the successor when the PCPP scores of the predecessor and successor are the same. This paper thinks the successor is closer to the potential crash point and has a higher probability of reaching it. Eventually, the PCPP high score node count is not very large.

Algorithm 2: Scoring the potential crash point's predecessor.

Input: The previously identified potential crash point list *PotentialCrashPointList*
Result: The scores of potential crash point's predecessor list *CurrentPCPPScorelist*

```
1  Worklist ← {PotentialCrashPointList};
2  for CurrentBasicBlockonCFG in Worklist do
3      CurrentPCPScorelist ← GetPCPScorelist(CurrentBasicBlockonCFG);
4      CurrentPCPPScorelist ← GetPCPPScorelist(CurrentBasicBlockonCFG);
5      CurrentPCPHashlist ← GetHashlist(CurrentBasicBlockonCFG);
6      if Countlist(CurrentPCPScorelist) > 0 then
7          if Hash(CurrentBasicBlockonCFG) not in CurrentPCPHashlist then
8              CurrentPCPPScorelist ← CurrentPCPPScorelist ∪ CurrentPCPScorelist;
9              CurrentPCPHashlist ← CurrentPCPHashlist ∪
                Hash(CurrentBasicBlockonCFG);
10             SetHashlist(CurrentPCPHashlist);
11         end
12     end
13     Predecessors ← GetPredecessoronCFG(CurrentBasicBlockonCFG);
14     for OnePredecessor in Predecessors do
15         PredecessorPCPPScorelist ← GetPCPPScorelist(OnePredecessor);
16         PredecessorPCPHashlist ← GetHashlist(OnePredecessor);
17         for CurrentPCPPScore, CurrentPCPHash in CurrentPCPPScorelist,
            CurrentPCPHashlist do
18             if CurrentPCPHash not in PredecessorPCPHashlist then
19                 PredecessorPCPPScorelist ← PredecessorPCPPScorelist ∪
                    CurrentPCPPScore;
20                 PredecessorPCPHashlist ← PredecessorPCPHashlist ∪
                    CurrentPCPHash;
21                 SetHashlist(PredecessorPCPHashlist);
22             end
23         end
24     end
25     Worklist ← Worklist ∪ Predecessors;
26 end
```

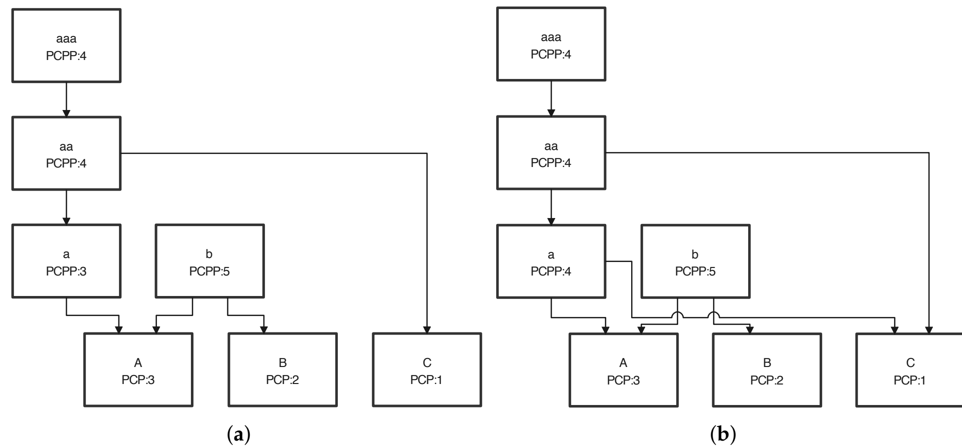


Figure 4. Scoring PCPP. (**a**) Scoring the potential crash point's predecessor. (**b**) Scoring the potential crash point's predecessor with reduplicate PCP.

(3) Scoring the potential error point

When ObFuzzer finds a potential crash point of the program and calculates a high score predecessor of the potential crash point, this does not mean that ObFuzzer already found the error and triggered the crash. ObFuzzer also needs to find a specified path from the high score predecessor to the potential crash point. This specified path is the way that can cause the crash.

A more complex path with more object operations will more likely trigger object-oriented program errors. Such a path is more effective than the sample path, which has fewer object operations. In the traditional procedure-oriented vulnerability-discovery technique, paths through more basic blocks or edges between basic blocks are considered more complex. Our object-oriented vulnerability-discovery techniques consider paths with more object operations more complex. Therefore, ObFuzzer needs to score the complexity of the basic blocks based on the object operation. If the path contains more complex (based on the object operation) basic blocks, the path is more complex. ObFuzzer tends to guide the program to execute more complex (based on the object operation) basic blocks during testing.

We can see from Algorithm 3 that when ObFuzzer scores a potential error point, ObFuzzer will obtain all points in the basic block that contain PCP scores and PCPP scores. These points represent the points related to potential crash points. Then, ObFuzzer will calculate the current PEP score of the basic block by the number of the predecessor nodes of these points in the data dependency graph. The nodes operate more data that are related to potential crash points' objects and come from different predecessor nodes. It has a higher score. This paper thinks that the higher the PEP score, the more probable it is that the basic blocks will generate errors during execution and this score will be used in the fuzzing phase to guide the program on which basic blocks to prefer to execute.

Algorithm 3: Scoring the potential crash point.

```
Input: Each basic block BasicBlock
Result: The score of potential error point PEPScore
1  PEPScore ← 0;
2  CurrentPCPlistonDDG ← GetCurrentPCPlistonDDG(BasicBlock);
3  CurrentPCPPlistonDDG ← GetCurrentPCPPlistonDDG(BasicBlock);
4  for CurrentPCPonDDG in CurrentPCPlistonDDG do
5  |  PCPPredecessors ← GetPredecessoronDDG(CurrentPCPonDDG);
6  |  PEPScore ← PEPScore + CountPredecessors(PCPPredecessors);
7  end
8  for CurrentPCPPonDDG in CurrentPCPPlistonDDG do
9  |  PCPPPredecessors ← GetPredecessoronDDG(CurrentPCPPonDDG);
10 |  PEPScore ← PEPScore + CountPredecessors(PCPPPredecessors);
11 end
```

As shown in Figure 5, the object-related (potential crash point's object) data operated by node "a" comes from three predecessors and the object-related data operated by node "b" comes from one predecessor (data dependency graph). Fuzzing to node "a" has more complex data operations related to the potential crash point's object than fuzzing to node "b" and is more likely to trigger an error in the program. Therefore, node "a" should have a higher potential error points score than node "b". We call this score the PEP (potential error points) score.

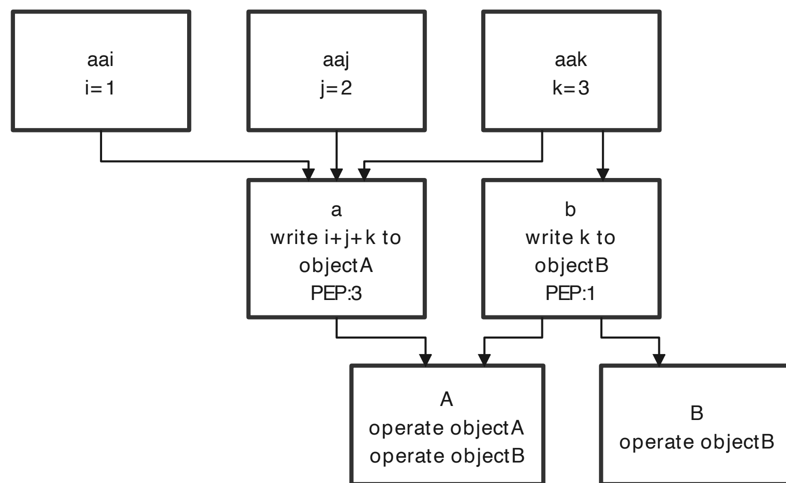


Figure 5. Scoring the potential error point.

4.4. Symbolic Execution: Guiding Fuzzing to Object Operation Related Positions

In order to implement an object-oriented approach to vulnerability discovery, we propose two dynamic test case generation approaches and propose ExeTree to assist ObFuzzer in path planning.

The PCPP score represents the correlation between the current basic block and the potential crash point. A basic block with a high PCPP score represents a large number of high-risk potential crash points in the successor of this basic block. ObFuzzer wants to start fuzzing at a position with a high correlation with potential crash points instead of letting fuzzing get stuck with a lot of non-crash-related security checks or error handling.

Therefore, ObFuzzer needs symbolic execution to assist it in guiding the program to the specific basic block, which has a high PCPP score.

ObFuzzer provides two methods of test case generation based on symbolic execution: high-performance test case generation and high-speed test case generation.

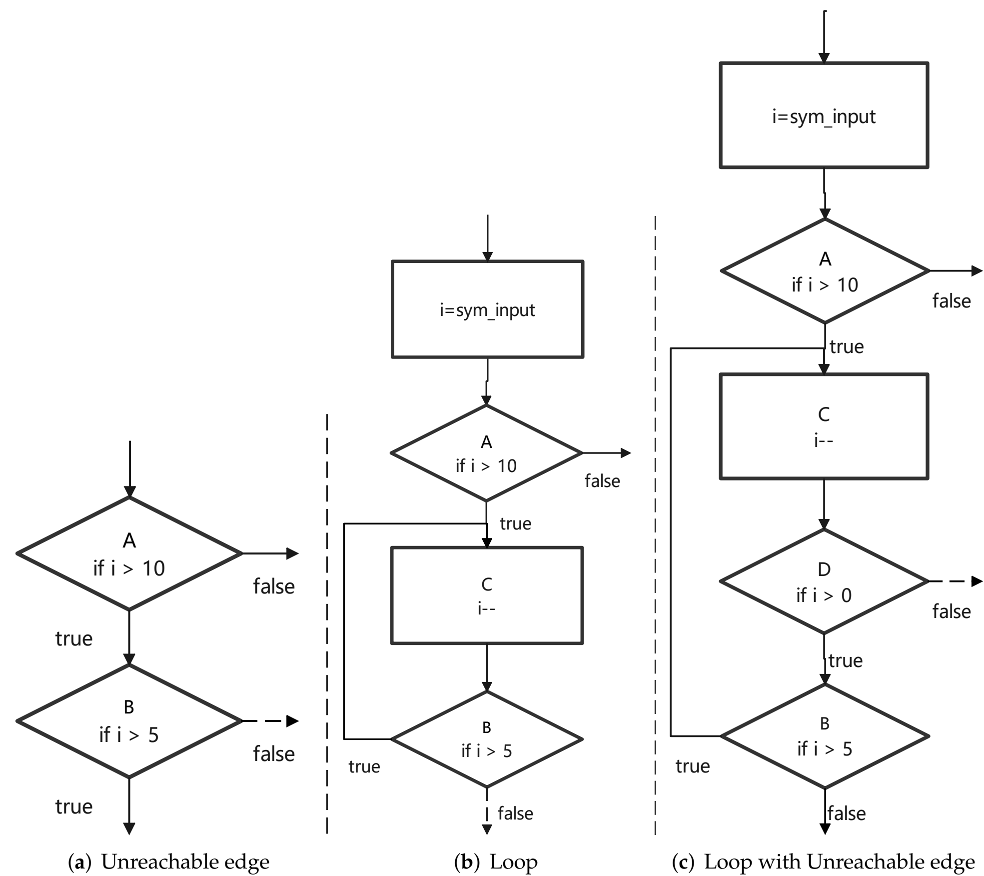
4.4.1. High-Performance Test Case Generation

High-performance test case generation focuses on performance. It tries to explore the specific basic blocks that are hidden deep in the program. Since global variables are often used, high-performance test case generation can only use a single core, which greatly affects execution speed.

Our high-performance test case generation is divided into three parts: accessibility of path, path planning and generate test cases.

(1) Accessibility of Path

Our ultimate goal is to obtain test cases for a basic block that has a higher PCPP score. However, all of our previous information was obtained from static analysis. ObFuzzer needs to convert this static data into dynamic symbolic execution states in order to generate the corresponding test cases using the satisfiability solver. Therefore, ObFuzzer needs to direct the symbolic execution engine to the basic block that has a higher PCPP score. ObFuzzer can easily find a path to the target basic block from the control flow graph. However, this path is often unreachable in dynamic execution. This makes it difficult for ObFuzzer to generate the test case for the target basic block. We illustrate this problem with Figure 6.



(**a**) Unreachable edge (**b**) Loop (**c**) Loop with Unreachable edge

Figure 6. Accessibility of Path.

As shown in Figure 6a, when ObFuzzer obtains a path from the control flow graph that requires a program to execute the branch that the false condition of the basic block “B” points to, this path is unreachable. Because “i” needs to be greater than 10 to execute the basic block “B”, there is no case where “i” is not greater than 5. This situation makes it difficult to generate a path from the control flow graph that can reach the target basic block from the entry point.

One of the most straightforward ideas to solve this problem is deleting such unreachable edges from the control flow graph whenever they are encountered. The accuracy of the control flow graph is improved by deleting such unreachable edges, which improves the accuracy of the path generated by the control flow graph. However, in reality, things are often not that simple.

As shown in Figure 6b, when ObFuzzer executes into a loop, there are few exits when the loop is executed only once. If ObFuzzer follows the above approach and finds that the false condition of node “B” cannot be satisfied when the program first executes the loop body, ObFuzzer deletes this edge from the control flow graph. This deletes an edge from the control flow graph that could have been executed, which seriously affects the accuracy of the control flow graph. This makes the control flow graph unable to assist ObFuzzer in generating a path to the target basic block.

A new idea can solve this problem by dividing the unreachable edges into in-loop unreachable edges and out-loop unreachable edges. Out-loop unreachable edges will be deleted and in-loop unreachable edges will continue to be executed. However, in reality, we often encounter more complex situations.

As shown in Figure 6c, when the unreachable edges of Figure 6a are located in the loop of Figure 6b, our previous proposed approach fails again. It is difficult to determine whether the false edge of node “D” should be deleted. In object-oriented programs, this

repeated code execution is not only a simple loop but also a recursion or other forms of code reuse. This makes path-accessibility analysis more difficult.

In practice, ObFuzzer can only do its best to determine whether an edge is reachable or not. That is, ObFuzzer analyzes the unreachable edges in the repeated code. When certain conditions that ObFuzzer sets are reached, ObFuzzer deletes the unreachable edges from the control flow graph and considers them completely unreachable.

To better solve the reachability issue, ObFuzzer proposes a new data structure, the ExeTree. With the ExeTree, ObFuzzer can transform the control flow graph of static analysis into a tree structure of dynamic execution. It can more easily assist ObFuzzer in obtaining a path from the entry point to the target basic block. ObFuzzer runs the program from the entry point by symbolic execution. ObFuzzer records the basic block of the program as a node and the successor basic block that the program can execute after this basic block is executed as the leaf node of the original node. Iterate this operation to build a binary tree. Therefore, the unreachable edge in Figure 6a cannot be added to the ExeTree.

To solve the reachability in loop issue, ObFuzzer sets two thresholds, NoBA (number of basic analyses, more than 2) and NoUA (number of unreachable analyses). NoBA represents the maximum number of times a basic block can be analyzed if it is out of a loop or if all the edges in the loop are reachable. This ensures that a basic block will not be analyzed indefinitely during symbolic execution and that symbolic execution will necessarily terminate. NoUA represents the maximum number of times a basic block can be analyzed if it is in a loop and has unreachable edges. When there are unreachable edges in the loop, ObFuzzer tries to analyze the loop body more times to try to change the accessibility of the edges.

There exists a situation like Figure 6c. When it is difficult to determine whether an edge in the loop is a definite unreachable edge, the basic block is only analyzed NoUA times and would not add to the ExeTree. Ensure that no more pointless loops are made for extensive analysis.

(2) Path Planning

Once the ExeTree of the program is obtained, ObFuzzer can plan the execution path of the program and guide the program to the basic block, which has a high PCPP score.

ObFuzzer starts at the entry point and searches the ExeTree in a breadth-first manner. Whenever ObFuzzer analyzes a basic block, it creates a PathNode object. In the PathNode object, there is a pointer storing the predecessor node of the current node. When ObFuzzer analyzes the target basic block, it adds the PathNode of the target basic block to a found list. After that, ObFuzzer takes the PathNode in the found list and tracks its chain of predecessors. Then, ObFuzzer can obtain a complete path from the entry point to the target basic block.

ObFuzzer can construct a binary tree like the one in Figure 7. When the basic block “D” is our target basic block, ObFuzzer will find the basic block “D” in the found list and follow the pointer of “D” to iteratively obtain its predecessor node. Thus, a complete path is constructed from the entry point to the target basic block “D”.

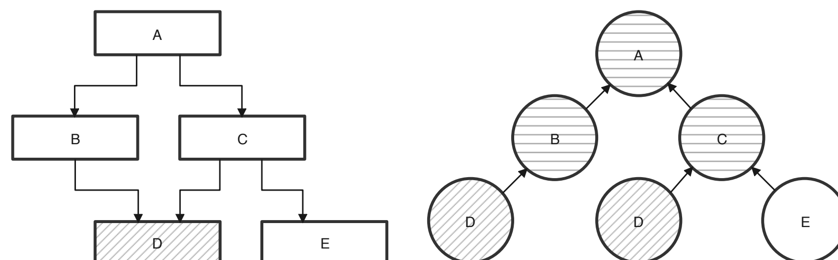


Figure 7. CFG to PathTree.

(3) Generating Test Cases

ObFuzzer starts executing the program from the entry point of the program. For each basic block encountered during execution, ObFuzzer compares its successor nodes with the path-planning information obtained in the previous step. The successors that are different from our planned path will be pruned and only the successors that are relevant to the plan will be retained for further execution. When the program executes to the target position, ObFuzzer stores the status information of the symbolic execution engine and obtains the corresponding test cases by satisfiability solver.

4.4.2. High-Speed Test Case Generation

The high-performance test case generation can effectively generate test cases that reach the specific PCPP high score basic block. However, in practice, this test case generation method has two shortcomings. First, the symbolic execution phase can only run on a single core, severely limiting the symbolic execution's efficiency. On the other hand, before generating test cases, ObFuzzer needs to set two thresholds (NoBA, NoUA). Different programs often have different thresholds that are appropriate. To generate more effective test cases, prior knowledge is often needed to assist in determining appropriate thresholds. This prior knowledge often requires manual assistance, which can seriously affect the performance of automated tests. Therefore, ObFuzzer proposes an optional alternative to implement test case generation to solve these problems. This approach allows efficient parallel generation of test cases without thresholds.

The high-speed test case generation approach explores the program's execution state through a symbolic execution engine. Then, each state of the program is given an independent symbolic execution engine. This allows them to be analyzed in different cores. Eventually, the termination condition of the analysis is decided based on the system's memory usage. Test cases are generated when a PCPP high score basic block is encountered during exploration.

This high-speed test case generation method can improve the efficiency of test case generation. However, due to the absence of path planning, a large number of meaningless paths are also analyzed and analysis resources are wasted. The test cases generated in this way often only capture the lighter PCPP high score basic block of the program, making it difficult to reach the deeper parts of the program. The ultimate purpose of the test cases is to guide the program execution to the PCPP high score basic block. Therefore, the depth of the PCPP high score basic block has little effect on the fuzzing phase. This method is also an option for test case generation.

4.5. Fuzzing: Randomly Combining Object Operation Order to Test the Target Program

In order to implement an object-oriented approach to vulnerability discovery, we propose the fuzzing approach based on the guide by object operation complexity. By converting the traditional coverage-based guide approach into an object operation complexity-based guide approach, we have completed the conversion from procedure-oriented vulnerability discovery to object-oriented vulnerability discovery.

After completing the static analysis and symbolic execution, ObFuzzer obtained the score of object operation complexity (PEP score) for each basic block and the test cases which could guide the program to the basic block with a high PCPP score. The Fuzzing phase accomplishes the following three main functions.

First, ObFuzzer needs to guide the program to the basic block, which has a high PCPP score. Among the procedure-oriented fuzzer, each edge is meaningful because the new edge is the base for the coverage-guided fuzzing. Most of the basic blocks that will be executed to are contained by at least two edges. Therefore, in a procedure-oriented fuzzer, it is meaningful to test for every basic block. Unlike the procedure-oriented fuzzer, the object-oriented fuzzer is guided based on the size of the object operation complexity. A larger object operation complexity means that the test case has a greater number of object operations during execution. It is well known that the more an object is operated on, the

more difficult it is to guarantee its security. Especially in large programs, an object stored in the heap may be used in many locations in the program. Even if the code is not done by one person, the security of the objects operated by such a code is more difficult to guarantee. This paper believes that the order of object operations found by the object-oriented sort is more dangerous than the order of basic block execution found by the procedure-oriented sort. Not every basic block will operate an object, having object operation complexity. This is especially significant in the early stages of program execution before a large number of objects are created. At this stage, there are often a large number of security checking operations that prevent test cases from exploring the core function of the program where a large number of object operations exist. Therefore, ObFuzzer needs test cases generated during the symbolic execution phase to assist it in bypassing the security checks and reaching the core part of the program.

Next, this paper proposes a new object operation complexity-based guide approach to convert the traditional procedure-oriented fuzzer into an object-oriented fuzzer.

Finally, ObFuzzer screens and stores the seeds. Then, the more efficient seeds have a higher probability of being selected in the next round of tests.

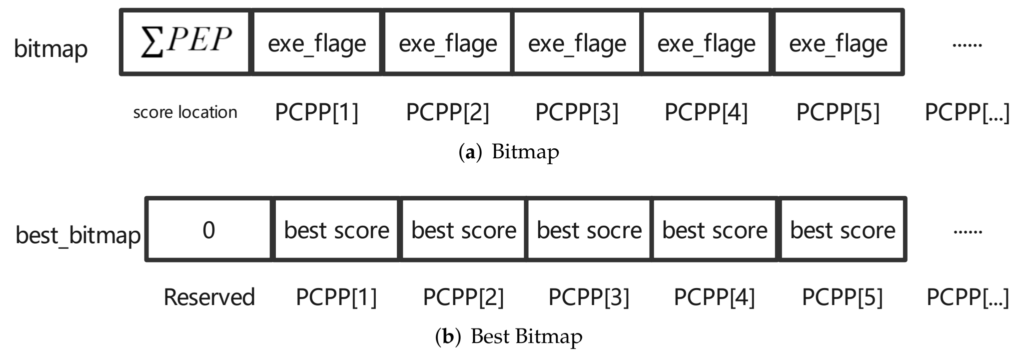
4.5.1. Guide Program Execution Path

During the symbolic execution phase, ObFuzzer generates test cases that can reach the target basic block in which the PCPP score is high. ObFuzzer has found in practice that the meaningful part of the test cases is often a set of binary data or a set of strings. Therefore, the mutation and testing of other parts of the test case have no significant improvement in object operation complexity. In practice, ObFuzzer extracts the critical information from the test cases to form a dictionary. The data in the dictionary are randomly spliced and inserted into the havoc phase of fuzzing to bypass the extensive security checks at the beginning of the program and get directly to the specific basic block that ObFuzzer is interested in. A better result is often obtained by computing the complexity of object operations after the target basic block. With the assistance of this guide, our fuzzer wastes little time on positions such as security checks that ObFuzzer does not care about, focusing more on exploring object operations after the PCPP high score basic block.

4.5.2. Calculating Path Complexity

ObFuzzer calculates path complexity according to the following steps. First, the object operation complexity score (PEP score) is accumulatively recorded for each basic block. The same basic block is repeatedly accumulated so that complex object operations embedded in the loop can be efficiently explored. Second, ObFuzzer records which PCPP high score basic blocks have been executed for each test case. Because the PCPP score represents the relationship between each basic block and potential crash point (PCP). ObFuzzer needs to observe the impact of different test cases on potential crash points by recording each execution of PCPP high score basic blocks. Eventually, ObFuzzer will summarize the above two types of information in the same array and analyze the program's execution situation. Position 0 of the array is used to accumulate the object operation complexity of the test cases and the subsequent positions of the array are used to mark the execution of the PCPP high score positions. ObFuzzer can simply analogize this array to the coverage bitmap in traditional procedure-oriented vulnerability discovery.

As shown in Figure 8, in the bitmap, position 0 (object operation complexity score position) stores the sum of the PEP scores of the basic blocks executed by the current test case. The subsequent position stores the execution flag of the corresponding PCPP high score basic blocks (like PCPP[1], PCPP[2], PCPP[3], etc.). The "exe flag" has only 0 or 1, two kinds of values; 0 represents not executed, 1 represents executed. In the "best bitmap", the value stored in position 0 (reserved) is fixed to 0. The subsequent position stores the maximum object operation complexity score (sum of PEP scores) for the corresponding PCPP high score basic block.



(**a**) Bitmap

(**b**) Best Bitmap

Figure 8. Format, Compare and Update of Bitmap.

When comparing "bitmap" and "best bitmap", ObFuzzer simply multiplies the value stored in "score position" in "bitmap" with the "exe flag" held in "PCPP" and then compares it with the "best score" stored in "PCPP" at the corresponding position in "best bitmap". If the value in the "bitmap" is larger, there is a test case that has executed to the current PCPP high score basic block with higher object operation complexity. The current test case has more complex object operations than the previously stored seeds. Therefore, the data stored in the current PCPP high score basic block position in the "best bitmap" should be replaced by the data stored in the "score position" in the current "bitmap". If the value in "best bitmap" is larger, it means that the previously executed test case has higher object operation complexity when this PCPP high score basic block has been executed. The current test case has less complex object operations than the previously stored seeds. Therefore, the update operation is not required. ObFuzzer uses this approach to determine whether a test case has found more complex object operations.

4.5.3. Sorting the Seeds

In the process of seed selection and storage, our first task is to evaluate the best test cases. ObFuzzer will use an array to record the highest object operation complexity of each PCPP high score basic block for the current fuzzer state. Each position in this recording array corresponds to a PCPP high score basic block and stores information about the highest object operation complexity test cases that can execute to the current PCPP high score basic block. With this array, ObFuzzer can quickly query and update the most complex object operation test cases for each PCPP high score basic block. These test cases will be identified as favorite seeds and have a higher priority in the subsequent fuzzing process.

As shown in Figure 9, in the "top map", the data stored in position 0 is NULL to be consistent with the "bitmap". The "top map" will query each test case by "PCPP", whether this test case executes to the current PCPP high score basic block. Among the results, the one with the highest complexity of object operations is selected as the favorite seed and its pointer is stored in the corresponding position of the "top map". ObFuzzer uses this approach to maintain an optimal seed sequence to rank the priority of different test cases in fuzzing.

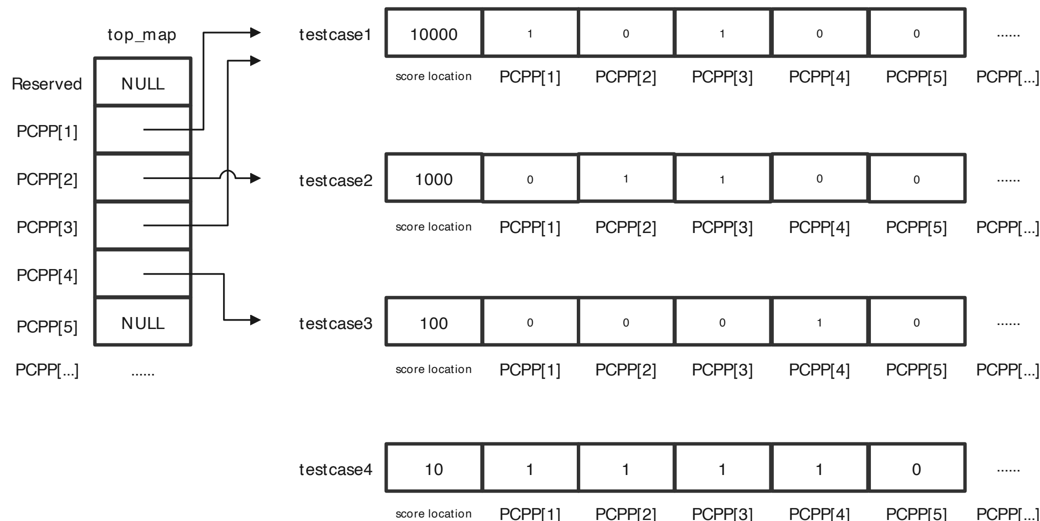


Figure 9. Top map.

5. Implementation

ObFuzzer implements static analysis and symbolic execution with Python and implements fuzzing with C.

5.1. Static Analysis

Our static analysis module is implemented through Angr [24], Joern [25] and CPG of the program [26]. Angr assists ObFuzzer in drawing the control flow graph (CFG) of the program in the mode of the emulated execution and Joern assists ObFuzzer in drawing the data dependency graph (DDG). In the Angr emulated module, we hook the memory-scheduling module of the simulator. In this way, Angr can obtain memory access information during program simulation execution while generating a control flow graph. Through further analysis of this memory access information, ObFuzzer finally obtains the potential crash point information (heap memory read, write and execute operations). Finally, ObFuzzer scores the program's PCP, PCPP and PEP.

5.2. Symbolic Execution

Our symbolic execution module is also implemented through Angr and Z3 Solver [27]. Angr helps ObFuzzer with the ability to simulate program execution and gather constraint information and Z3 Solver allows ObFuzzer to obtain solutions from constraints. When the program execution reaches a basic block with a relatively high PCPP score, ObFuzzer stores the execution state and generates the corresponding test cases. Then, ObFuzzer extracts the critical information from the test cases to form a dictionary and passes it to the fuzzing. By this dictionary, fuzzing can test the target position near a predecessor node with more relations to the potential crash point.

5.3. Fuzzing

Our fuzzing module is implemented based on AFL [28] and Honeybee [29]. AFL's path information-acquisition method adopts static instrumentation technology or QEMU. The basic block is identified by giving it a random number by identifying the basic block header and jump. Object-oriented vulnerability discovery also needs basic block information, which is used to identify the object operation complexity of basic blocks. It is important to determine which basic blocks are better at discovering vulnerabilities related to complex object operations. AFL's method of identifying basic blocks by random numbers is difficult to relate to the basic block information obtained by static analysis. Therefore, ObFuzzer uses Intel Processor Trace technology, which can easily link the dynamic execution address of the program (close address space layout randomization) with the static analysis address. At the same time, it extends the applicability of ObFuzzer to test binary-only programs.

The static analysis, symbolic execution and fuzzing modules do not require source code. ObFuzzer chooses Honeybee as our Intel Processor Trace component. Finally, ObFuzzer imports the execution address information into our modified object-oriented fuzzer.

6. Evaluation

This paper runs all our evaluations on 64-bit machines running Linux Mint 20.2 with Intel i9-8950HK CPUs (six cores in total) and 32 G memory. Every experiment was tested five times for 24 h.

Our evaluation aims to answer the following questions.

1. Performance overhead: How much performance overhead does ObFuzzer introduce compared to traditional procedure-oriented vulnerability discovery?

2. Object operation complexity improvement: How much improvement does ObFuzzer's object operation complexity provide compared to traditional procedure-oriented vulnerability discovery?

3. Test case generation efficiency: How much improvement is there in ObFuzzer's object operation complexity with the generated test cases?

4. Crash discovery: How many crashes did ObFuzzer find in the real program?

ObFuzzer proposed a new metric to guide the program to discover vulnerabilities. AFL is not fundamentally different from other coverage-based guided vulnerability-discovery tools under the new metric. Therefore, this uses AFL as the baseline in this paper in order to compare the performance overhead introduced by Intel Processor Trace and object-oriented metrics. This paper modified AFL by converting its traditional instrumentation-based information-acquisition method into an Intel Processor Trace-based information-acquisition method. Since the Intel Processor Trace is implemented through Honeybee, we call this vulnerability-discovery tool AFL-Honeybee. AFL-Honeybee is precisely the same as AFL except for the means of information acquisition and the corresponding basic block ID generation.

We used some real-world programs as the test set for this paper. We used "nm" and "strings" to represent procedure-oriented programs; "woff2 compress" and "woff2 decompress" to represent procedure-oriented programs, which incorporate complex data operations, "pdfdetach", "pdffonts", "pdfinfo", "pdftops", "pdftotext", to represent object-oriented programs. By testing different types of programs, we can better understand ObFuzzer's effect on different types of programs. This paper tested in parallel with a group of six fuzzing instances during the experiments.

6.1. Performance Overhead

ObFuzzer is divided into three parts. The static analysis and symbolic execution parts are executed only once. In contrast, the fuzzing part requires a lot of repeated executions to discover potential vulnerabilities in the program. Therefore, the execution efficiency of the fuzzing phase has a more significant impact on the effectiveness of vulnerability discovery. The efficiency of ObFuzzer's fuzzing phase execution is tested in this paper.

Since the execution information acquisition of AFL is implemented by program instrumentation, the execution information acquisition of ObFuzzer is implemented by Intel Processor Trace. Exploiting program vulnerabilities in the absence of source code is also an advantage of this paper. Therefore, it is not fair to directly compare the implementation efficiency of the two tools.

In practice, this paper adopts a multi-level comparison approach to more scientifically measure the execution efficiency of ObFuzzer. By comparing the AFL's execution efficiency with the AFL-Honeybee, we can obtain the performance loss introduced by Intel Processor Trace. Then, we compare AFL-Honeybee's execution efficiency with the ObFuzzer fuzzing phase and we can compare the performance difference between the coverage-based vulnerability-discovery approach and object operation complexity-based vulnerability-discovery approach. We present the execution efficiency of AFL, AFL-Honeybee and ObFuzzer fuzzing phase in Figures 10 and 11.

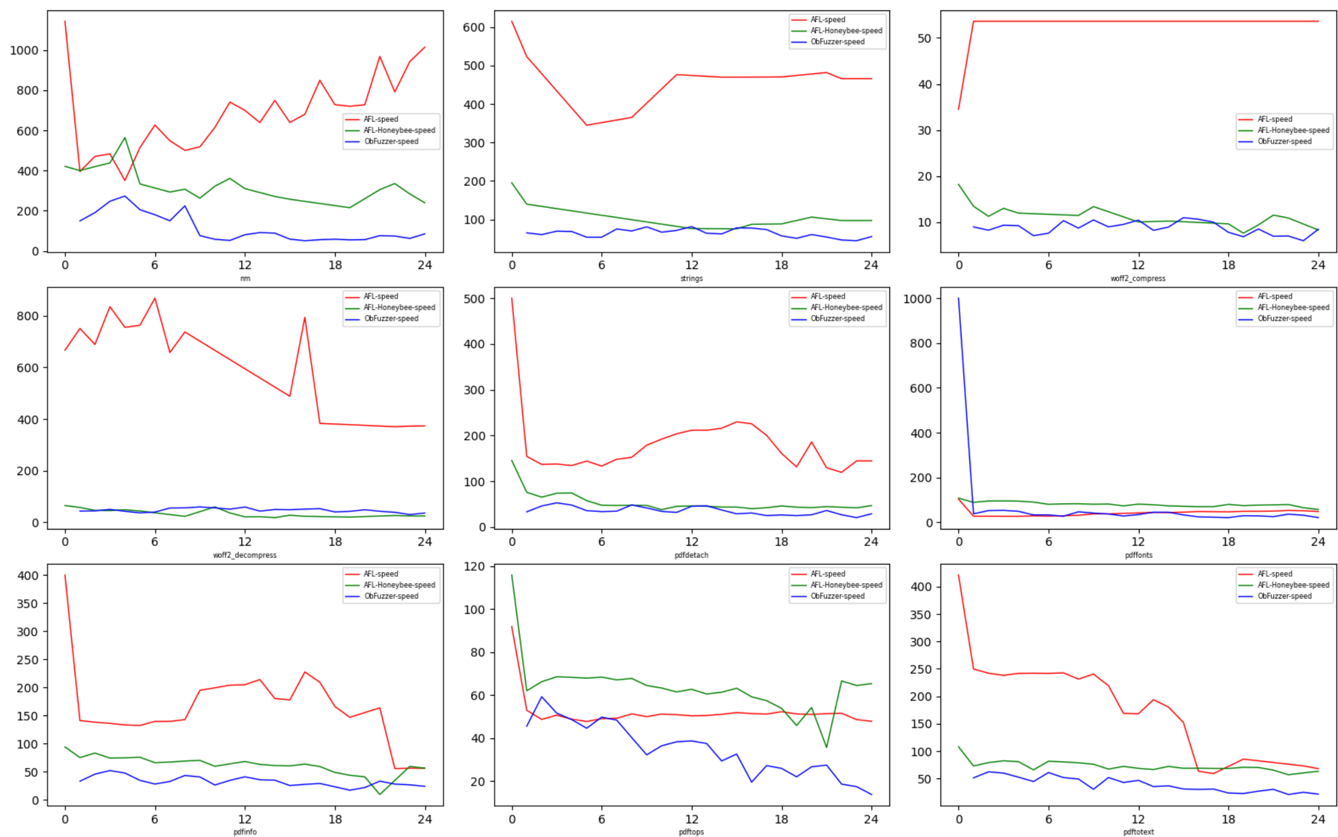


Figure 10. Execution Speed of AFL, AFL-Honeybee and ObFuzzer Fuzzing Phase.

In Figure 10, the x-axis represents test time (hours) and the y-axis represents the number of executions per second. The red line represents the execution efficiency of AFL, the green line represents the execution efficiency of AFL-Honeybee and the blue line represents the execution efficiency of ObFuzzer fuzzing phase. Experimentation shows that a simple Intel Processor Trace approach to information acquisition can introduce significant overhead when testing procedure-oriented programs. This conclusion can be drawn from a comparison of the red and green lines in the first five plots in Figure 10, which is a comparison of AFL and AFL-Honeybee. The overhead of converting procedure-oriented metrics to object-oriented metrics on the base of Intel Processor Trace is relatively small. This conclusion can be drawn from a comparison of the green and blue lines in the first five plots in Figure 10, which is a comparison of AFL-Honeybee and the ObFuzzer fuzzing phase. In object-oriented programs, the overhead introduced by Intel Processor Trace is further reduced. This conclusion can be drawn from a comparison of the green and blue lines in the last four plots in Figure 10, which is a comparison of AFL-Honeybee and the ObFuzzer fuzzing phase. The main reason for this is that more instructions are executed in object-oriented programs, which can better apportion the performance loss. In addition, due to the slow execution efficiency of "woff2 compress", with AFL it is hard to find any new information and AFL cannot output execution efficiency information. Therefore, the plotted execution efficiency is approximately linear.

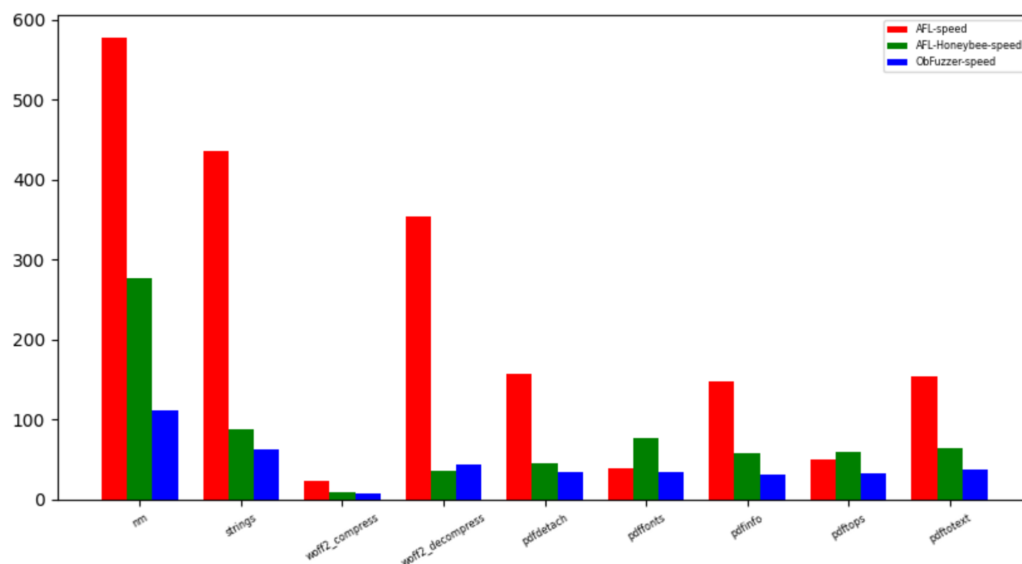


Figure 11. Execution Speed Average of AFL, AFL-Honeybee and ObFuzzer Fuzzing Phase.

In Figure 11, the x-axis represents different programs and the y-axis represents the number of executions per second. The red bar represents the number of executions per second of AFL, the green bar represents the number of executions per second of AFL-Honeybee and the blue bar represents the number of executions per second of the ObFuzzer fuzzing phase. Intel Processor Trace introduces 70% average performance overhead in procedure-oriented programs compared with AFL. Moreover, it introduces 14% average performance overhead in object-oriented programs. Object-oriented metrics based on Intel Processor Trace introduce 21% average performance overhead in procedure-oriented programs compared with AFL. Furthermore, it introduces 42% average performance overhead in object-oriented programs. Object-oriented programs require more data to be tracked and analyzed by ObFuzzer. Therefore, the object-oriented metrics introduce more performance overhead in object-oriented programs.

Compared with AFL, a significant performance overhead occurs with the introduction of Intel Processor Trace. At the same time, the object-oriented metrics approach introduces a smaller performance overhead within acceptable limits.

6.2. Object Operation Complexity Improvement

ObFuzzer is fundamentally an object-oriented vulnerability-discovery tool. Object operation complexity is the core guide approach. More complex object operations are more likely to trigger object-related vulnerabilities. Therefore, we need to compare the differences between traditional procedure-oriented vulnerability-discovery approaches and object-oriented vulnerability-discovery approaches regarding the complexity of object operations.

There is no object operation complexity-based metric in AFL. Therefore, obtaining the object operation complexity during AFL in real time is difficult. We put the seeds obtained during the AFL execution into the fuzzing phase of ObFuzzer. The object operation complexity during the execution of AFL is approximated. Then, we compare the execution results with ObFuzzer's fuzzing phase to further demonstrate the effectiveness of the object-oriented guide model.

As we can see from Table 1, we take the object operation complexity obtained with AFL-Honeybee as the base and calculate the ratio of the object operation complexity obtained with ObFuzzer. The numbers in brackets represent the PCPP high score positions that ObFuzzer executes. The object operation complexity ratio for procedure-oriented programs is between 1 and 1.17. The object operation complexity ratio for object-oriented programs is between 1.29 and 1.40. ObFuzzer works better on object-oriented programs. In the

"pdfinfo" program, the object operation complexity obtained by ObFuzzer is significantly smaller than that obtained by AFL-Honeybee. This is because AFL-Honeybee randomly generates a test case with the probability of having a large object operation complexity, making ObFuzzer's object operation complexity look smaller.

Table 1. This table contains two parts. First, the object operation complexity of ObFuzzer with AFL-Honeybee as the base. Second, the numbers in brackets are the counts of PCPP high score positions executed in the program test.

	AFL-Honeybee	ObFuzzer
nm	1 (122)	1.17 (144)
strings	1 (37)	1.10 (37)
woff2 compress	1 (19)	1.00 (19)
woff2 decompress	1 (16)	1.00 (16)
pdfdetach	1 (346)	1.33 (463)
pdffonts	1 (420)	1.30 (426)
pdfinfo	1 (337)	0.13 (349)
pdftops	1 (347)	1.29 (367)
pdftotext	1 (333)	1.40 (495)

At the same time, ObFuzzer explored the PCPP high score position equal to AFL-Honeybee when testing procedure-oriented programs. ObFuzzer explored the PCPP high score position and it was found that it is significantly better than AFL-Honeybee when testing object-oriented programs.

6.3. Test Case Generation Efficiency

Through the above two experiments, we demonstrate that ObFuzzer can significantly improve the complexity of object operations in program fuzzing at a lower performance loss.

Next, we need to test whether the test cases generated by the symbolic execution phase can effectively increase the complexity of object operations in the fuzzing phase. This paper designed two sets of experiments. One is the object operation complexity obtained by ObFuzzer without the dictionary extracted in test cases; the other is the object operation complexity obtained by ObFuzzer with the dictionary extracted in test cases. By comparing the results, we can understand the impact of the test cases generated in the symbolic execution phase on the object operation complexity of ObFuzzer.

As we can see from Figure 12, the red line represents the object operation complexity without the dictionary and the blue line represents the object operation complexity with the dictionary. The experiments show that the maximum value of object operation complexity for procedure-oriented programs does not relate much to using the generated test cases. However, most of the time, ObFuzzer can find the object operation of more complex new test cases with the dictionary earlier. As shown in the "strings" program, in some cases, test cases with higher object operation complexity require a longer time for mutation and testing. As a result, the complexity of object operations with generated test cases is less in some periods than that of object operations without the dictionary. ObFuzzer performs relatively better in testing object-oriented programs with the dictionary. Both speed and object operation complexity is improved. Therefore, the test cases generated by the symbolic execution phase improve the complexity of object operations in the fuzzing phase.

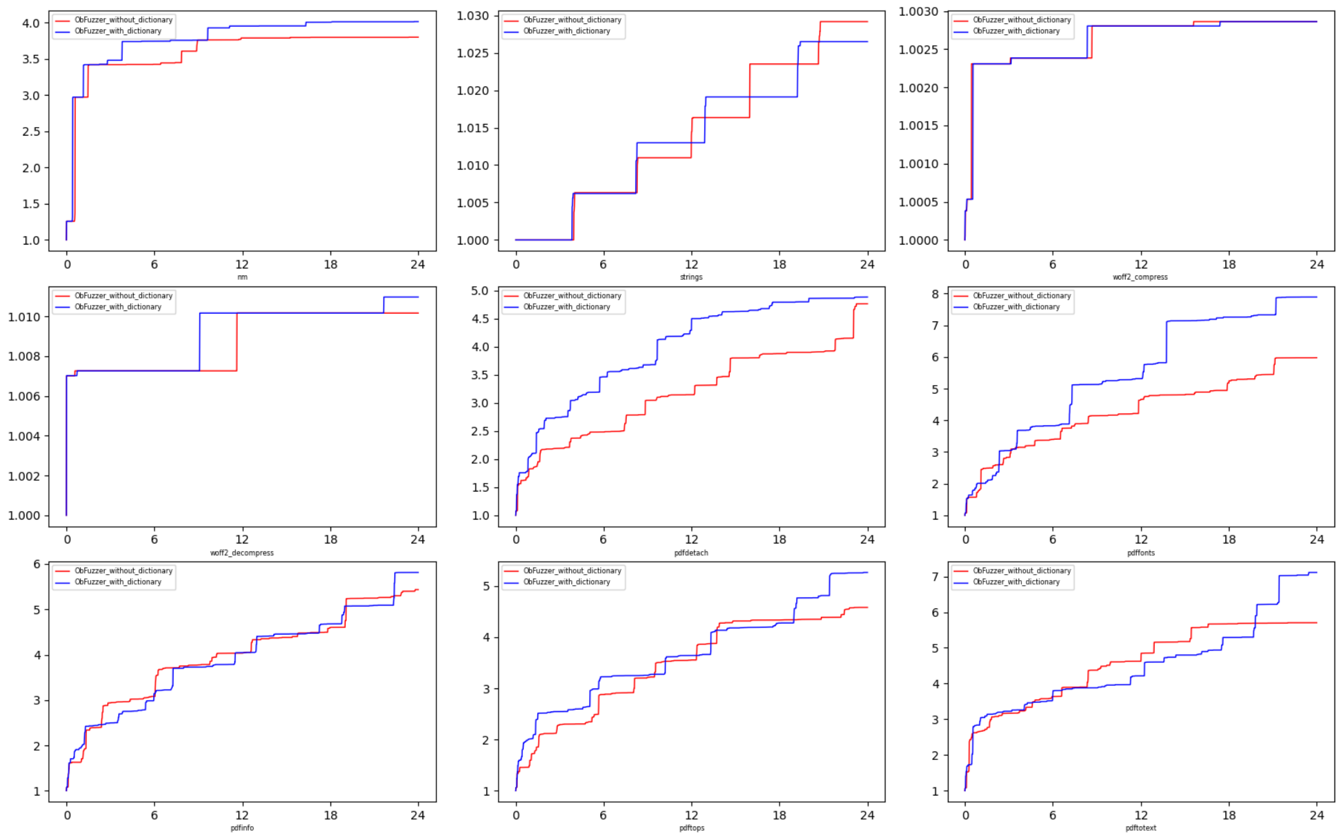


Figure 12. Object Operation Complexity of ObFuzzer with/without test case dictionary.

6.4. Crash Discovery

We compared the unique vulnerabilities and errors found by AFL and ObFuzzer.

As we can see from Table 2, ObFuzzer found no vulnerabilities with AFL, five unique vulnerabilities and one logic error without a crash with ObFuzzer. Through experiments, we can find that at this stage of real-world programs, it has been difficult to discover vulnerabilities with purely procedure-oriented AFL. Through the object-oriented approach to vulnerability discovery, ObFuzzer can find new vulnerabilities in real-world programs. This is the only result ObFuzzer found in testing "xpdf". Some of these crashes can make several "xpdf" program crashes. The experiment shows object-oriented vulnerability-discovery methods can obtain better results than traditional procedure-oriented vulnerability-discovery methods in complex object-oriented programs.

Table 2. The unique vulnerabilities and errors ObFuzzer found in AFL and ObFuzzer.

	AFL		ObFuzzer	
	Crash	**Error**	**Crash**	**Error**
nm	0	0	0	0
strings	0	0	0	0
woff2 compress	0	0	0	0
woff2 decompress	0	0	0	0
pdfdetach	0	0	0	3
pdffonts	0	0	0	1
pdfinfo	0	0	0	0
pdftops	0	0	0	1
pdftotext	0	0	1	0

7. Discussion

Our core work is to modify the traditional coverage-guided (procedure-oriented) vulnerability-discovery tools to an object-oriented hybrid fuzzer. This paper converts the traditional scheduling method from whether a new edge is found to whether a more complex object operation is found. The complexity of object operations is measured by the PEP score of basic blocks on the path. This produces the fundamental difference between object-oriented vulnerability-discovery technology and procedure-oriented vulnerability-discovery technology. Due to the need for execution efficiency, procedure-oriented vulnerability-discovery tools are often more inclined to select test cases with smaller storage space and higher execution efficiency under the same coverage. The object-oriented vulnerability-discovery tool is more inclined to more complex object operations and deeper path exploration. Vulnerabilities that hide deep in the program and require the same object to be operated many times are difficult to discover with traditional procedure-oriented vulnerability-discovery tools. The Object-oriented vulnerability-discovery tool is more effective in solving these types of problems.

This paper experimented with object-oriented vulnerability discovery and obtained the following conclusions.

1. Object-oriented metrics introduce less performance overhead;
2. The object-oriented guide approach can effectively improve the object operation complexity when testing the target program;
3. Test cases generated in the symbolic execution phase can assist in the fuzzing phase to improve the object operation complexity;
4. Object-oriented vulnerability-discovery methods can discover vulnerabilities that are difficult to discover with the procedure-oriented vulnerability-discovery methods caused by complex object operations.

Moreover, we can find that ObFuzzer on object-oriented programs for vulnerability-discovery effect is significantly better than the effect of procedure-oriented programs, even in cases where there is a 20% to 40% performance overhead in execution efficiency. We argue that the guide for object operation complexity is better in a practical vulnerability-discovery scene as the execution time grows. This is due to the procedure-oriented vulnerability-discovery method's coverage bitmap being gradually filled and the guidance ability gradually decreasing as the test time grows. This will make it difficult to discover new seeds. In contrast, in the object-oriented vulnerability-discovery method, object operation complexity is accumulative and the guidance ability will not decrease with the growth in the test time.

ObFuzzer is able to effect a more efficient exploration of complex object operations in vulnerability discovery. This gives a new perspective to the traditional coverage-based vulnerability-discovery techniques. At the same time, it proves that vulnerabilities caused by complex object operations are widespread in programs.

8. Conclusions

This paper proposes an object-oriented vulnerability-discovery approach. This approach effectively solves the disadvantages of traditional procedure-oriented vulnerability-discovery approaches in vulnerabilities caused by object operations. We implemented our proposed approach in ObFuzzer and achieved significant results in discovering complex object-oriented programs.

Author Contributions: Conceptualization, X.H.; Funding acquisition, P.W., K.L. and X.Z.; Investigation, X.H.; Methodology, X.H.; Software, X.H.; Supervision, P.W.; Validation, X.H.; Writing—original draft, X.H.; Writing—review and editing, X.H. and P.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Natural Science Foundation of China (61902412, 61902405, 62272472), the National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), the Natural Science Foundation of Hunan Province of China under Grant No. 2021JJ40692 and the Research Project of the National University of Defense Technology (ZK20-17, ZK20-09).

Data Availability Statement: Not applicable.

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

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

Abbreviations

The following abbreviations are used in this manuscript:

PCP	Potential Crash Point
PCPP	Potential Crash Point's Predecessor
PEP	Potential Error Point

References

1. Wang, P.; Zhou, X. SoK: The progress, challenges and perspectives of directed greybox fuzzing. *arXiv* **2020**, arXiv:2005.11907.
2. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. Collafl: Path sensitive fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 679–696.
3. Hsu, C.C.; Wu, C.Y.; Hsiao, H.C.; Huang, S.K. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In Proceedings of the Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research, San Diego, CA, USA, 18 February 2018.
4. Zhang, C.; Dong, W.Y.; Ren, Y.Z. INSTRCR: Lightweight instrumentation optimization based on coverage-guided fuzz testing. In Proceedings of the 2019 IEEE 2nd International Conference on Computer and Communication Engineering Technology (CCET), Beijing, China, 16–18 August 2019; pp. 74–78.
5. Nagy, S.; Nguyen-Tuong, A.; Hiser, J.D.; Davidson, J.W.; Hicks, M. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual Event, 11–13 August 2021; pp. 1683–1700.
6. Intel, L. Circumventing Fuzzing Roadblocks with Compiler Transformations. Available online: https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/ (accessed on 15 August 2016).
7. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 711–725.
8. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017; Volume 17, pp. 1–14.
9. Keung Luk, C. Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of the Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005; pp. 190–200.
10. Bellard, F. QEMU A generic and open source machine emulator and virtualizer. Available online: https://www.qemu.org/ (accessed on 4 February 2017).
11. Zhang, G.; Zhou, X.; Luo, Y.; Wu, X.; Min, E. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access* **2018**, *6*, 37302–37313. [CrossRef]
12. Wang, Y.; Wu, Z.; Wei, Q.; Wang, Q. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access* **2019**, *7*, 36340–36352. [CrossRef]
13. Österlund, S.; Razavi, K.; Bos, H.; Giuffrida, C. {ParmeSan}: Sanitizer-guided Greybox Fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Virtual Event, 12–14 August 2020; pp. 2289–2306.
14. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. {AddressSanitizer}: A Fast Address Sanity Checker. In Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), Boston, MA, USA, 13–15 June 2012; pp. 309–318.
15. Zaddach, J.; Bruno, L.; Francillon, A.; Balzarotti, D. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2014; Volume 14, pp. 1–16.
16. Muench, M.; Nisi, D.; Francillon, A.; Balzarotti, D. Avatar 2: A multi-target orchestration platform. In Proceedings of the Workshop Binary Analysis Research (Colocated NDSS Symposium), San Diego, CA, USA, 18 February 2018; Volume 18, pp. 1–11.
17. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Lin, Z.; Wang, X.; Lau, W.C.; Sun, M.; Yang, R.; Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT through App-based Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 18–21 February 2018.
18. Song, C.; Zhou, X.; Yin, Q.; He, X.; Zhang, H.; Lu, K. P-fuzz: A parallel grey-box fuzzing framework. *Appl. Sci.* **2019**, *9*, 5100. [CrossRef]
19. 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.
20. 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 (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
21. Huang, H.; Yao, P.; Wu, R.; Shi, Q.; Zhang, C. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1613–1627.

22. Chen, Y.; Li, P.; Xu, J.; Guo, S.; Zhou, R.; Zhang, Y.; Wei, T.; Lu, L. Savior: Towards bug-driven hybrid testing. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1580–1596.
23. Kim, K.; Jeong, D.R.; Kim, C.H.; Jang, Y.; Shin, I.; Lee, B. HFL: Hybrid Fuzzing on the Linux Kernel. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2020.
24. Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; et al. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016.
25. Joern—The Bug Hunter's Workbench. Available online: https://github.com/joernio/joern (accessed on 14 March 2019).
26. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; pp. 590–604.
27. Moura, L.D.; Bjørner, N. Z3: An efficient SMT solver. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 29 March–6 April 2008; Springer: Berlin, Germany, 2008; pp. 337–340.
28. Zalewski, M. American Fuzzy Lop. Available online: https://github.com/google/AFL (accessed on 25 July 2019).
29. Husain, A.; Dinaburg, A.; Goodman, P. Honeybee. Available online: https://github.com/trailofbits/Honeybee (accessed on 21 January 2020).