applied sciences

Article

SHFuzz: Selective Hybrid Fuzzing with Branch Scheduling Based on Binary Instrumentation

Xianya Mi, Baosheng Wang *, Yong Tang, Pengfei Wang and Bo Yu

College of Computer, National University of Defense Technology, Changsha 410073, China; mixianya09@nudt.edu.cn (X.M.); ytang@nudt.edu.cn (Y.T.); pfwang@nudt.edu.cn (P.W.); yubo0615@nudt.edu.cn (B.Y.)

* Correspondence: bswang@nudt.edu.cn

Received: 13 July 2020; Accepted: 3 August 2020; Published: 6 August 2020

Abstract: Hybrid fuzzing is a popular software testing technique that combines random fuzzing with concolic execution. It is widely used in the security domain known for its ability to find deeply hidden vulnerabilities and reach high code coverage. Hybrid fuzzing is based on negating branches in the execution path of a specific input to generate new test cases. However, due to numerous inputs and related branches, it does not show the best of its effectiveness without input and branch selection methods. In this paper, we systematically analyze the branch scheduling problem in the internal attributes of hybrid fuzzing, focusing on the synchronization mechanism. To solve the problems, we propose the Selective Hybrid Fuzzing (SHF) approach with branch scheduling based on binary instrumentation. There are two major parts to the SHF approach: (1) we propose a critical branch selection algorithm to select critical branches by three metrics: hit accuracy, solvability, and complexity; (2) we propose a priority score calculation algorithm to select inputs by the number of critical branches. With the SHF approach, we choose only the branches that can be negated to generate new coverage, instead of repeatedly executing the same branches and generating duplicates of inputs. We implement a hybrid fuzzer called SHFuzz with our SHF approach and compare it with the state-of-the-art hybrid fuzzer QSYM. In the evaluation, SHFuzz outperforms QSYM in 20 real-world applications from the Google Fuzzer Test Suite and other program suites in a 12 h test. On average, SHFuzz achieves 8.40% more code coverage and 100 more unique crashes in each application. Our work also finds existing vulnerabilities 7.85× faster than QSYM. We also find new bugs by SHFuzz, which QSYM fails to find. Our evaluation shows that the selective hybrid fuzzing approach can reduce the number of branches executed in concolic execution, enhancing hybrid fuzzing on code coverage and bug finding capabilities.

Keywords: concolic execution; coverage-based fuzzing; hybrid fuzzing; fuzzing; software testing

1. Introduction

Fuzzing [1] is a software testing method that randomly mutates the inputs, trying to reach the target execution paths. It is widely used in the software security domain, especially in finding vulnerabilities and increasing program coverage. Random testing lacks efficiency since the search space of random mutation is too large to reach full coverage. If we try to fuzz a program with the input length as long as 10 bytes, the search space will be 2^{80} (10 bytes, each byte with eight bits). The problem can be even worse when the length grows longer. Therefore, different fuzzing methods based on various assumptions have come out to satisfy different goals. Despite the diverse methods and techniques, the core ideas of these approaches are similar: reduce the size of the search space and reach the target path as fast as possible.

Based on the method of mutating inputs, there are three different methods: mutation-based fuzzing, grammar-based fuzzing, and hybrid fuzzing. Mutation-based fuzzing [2–5] uses different heuristics based on program analysis to reduce the search space for efficiency. The disadvantage of this method is that it is difficult to pass complicated format checks. Grammar-based fuzzing [6–8] solves this problem to some extent by inferring the structure of the inputs to pass the input format checks. However, this method needs extra work to build input file grammar before fuzzing. In recent years, hybrid fuzzing [9–14] has been widely researched and discussed in the security domain, which utilizes the effectiveness of concolic execution [15,16], as well as the efficiency of fuzzing. Concolic execution is a mixed method of symbolic execution [17] and concrete execution, which builds logical constraints of branches in the program and generates new test cases by negating and solving the branches with the help of Satisfiability Modulo Theories (SMT) solvers [18]. The advantage of hybrid fuzzing is that for complicated checks that are hard to find by mutation-based fuzzing, concolic execution can solve those checks and make program execution deeper into the longer paths. Meanwhile, fuzzing can find shallow paths with simple checks quickly and compensate for the larger resources spent by concolic execution. Existing works [9,10,15,19] show that hybrid fuzzing can achieve more program execution coverage and find more bugs compared with mutation-based fuzzing.

There are three implementation methods of applying hybrid fuzzing: demand launch, optimal switch, and synchronization. Driller [9] and other works [15] used the demand launch strategy, which only triggers the concolic execution engine when fuzzing gets stuck. The core idea of this method seems reasonable since concolic execution is good at solving complicated checks, which fuzzing fails to pass; however, it is hard to define what a "stuck" situation looks like. The previous work [11] showed that for most evaluations using Driller, the concolic execution engine is never triggered in the limited time because fuzzing never reaches a "stuck" state. MDPC [19] uses the optimal switch method, which computes the costs of solving every branch by fuzzing and concolic execution, respectively, and then chooses the more economic method for each branch. DigFuzz [11] extends this method by a step forward to make optimal decisions more efficient. The demand launch method and optical switch method share one important feature in common: the fuzzer and the concolic execution engine are triggered sequentially instead of in parallel. It may seem practical with single-core CPUs, which have limited computation capability; however, with modern multi-core CPUs and large memory space, the power of parallel computation is not utilized enough. Thus, the third method by synchronization comes out, which runs the concolic execution engine in parallel with the fuzzer and synchronizes the newly generated test cases with the fuzzer from time to time. QSYM [10] is the state-of-the-art hybrid fuzzing work, which has a highly tailored concolic execution engine designed for fuzzing and a synchronization method with the AFL [2] fuzzer. It shows impressive efficiency in finding vulnerabilities and increasing program execution coverage because running concolic execution in parallel can generate much more new inputs than the other two methods.

Although hybrid fuzzing with the synchronization method seems like the most efficient way, for now, there are still numerous unsolved problems related to its mechanism. Firstly, there are too many candidate inputs for concolic execution. The fuzzing engine can generate numerous candidate inputs in the queue, mostly as many as thousands of inputs in 12 h, while at the same time, concolic execution can only analyze hundreds of inputs at most. For example, we test libjpeg-turbo-07-2017 in the Google Fuzzer Test Suite [20] with QSYM for 12 h and find that there are 4420 test cases generated in the AFL slave queue, while QSYM, which selects the candidate input to perform concolic execution from this queue, analyzes 483 files and generates 1444 new test cases. In this case, only 10% of the candidate inputs are analyzed. If we choose the candidate inputs randomly, the effectiveness of finding deeper paths cannot be optimized. The reason for this problem is that the time of concolic execution for one input is normally counted by minutes or even hours, and as a result, in a certain amount of time, not many inputs can be executed. Thus, we need a clever input selection method for concolic execution.

Secondly, there are many branches in one input. Since the theory of concolic execution is to negate the branches of executing one input to generate new test cases, solving every branch in the program will take too much time, especially when there are loops in the code. Besides, most candidate inputs share same branches, causing the overlap of newly generated test cases. This means there will be duplicates of inputs that guide execution to the same coverage, and the fuzzer will take extra time to analyze and synchronize those duplicates and cause unnecessary overhead, i.e., time and memory assumption.

Thirdly, different branches have different complexities, and some of them are not suitable for concolic execution to solve. Different complexities of branches result in different times to solve by concolic execution. Some of the branches are easy enough for the fuzzer to find, so the newly generated test cases from these branches may not be synchronized by the fuzzer because they have already been found by the fuzzer. In other words, concolic execution should focus more on the complicated branches than the simple ones. In summary, the ideal way to perform hybrid fuzzing with synchronization is that every new test case generated by concolic execution should guide the execution to new coverage, which fuzzing can hardly reach. Our approach intends to get as close to this aim as possible.

In this paper, we propose a Selective Hybrid Fuzzing (SHF) approach to solve these problems and enhance the performance of hybrid fuzzing, with two main algorithms. (1) The critical branch selection algorithm: We use three metrics, hit accuracy, solvability, and complexity, to describe the critical information provided by each branch. Specifically, we use dynamic binary instrumentation, a technique to analyze binary programs with instrumentation when running the program, to get the information of program execution, based on which we try to find out the branches that can guide the execution to the unexplored path, solvable to get new test cases and hard to be found by the fuzzer. (2) The priority score calculation algorithm: We choose the input with the most optimal branches as the candidate input for concolic execution based on the branch scheduling results from the previous method.

Our contributions:

- We systematically study the branch scheduling problem in hybrid fuzzing with the synchronization mechanism. We analyze the three internal causes of the problem and try to solve them by proposing a novel solution based on binary instrumentation without the demand of source code.
- We propose the Selective Hybrid Fuzzing (SHF) approach with two algorithms: (1) the critical branch selection algorithm to calculate the critical score of each branch by three metrics: hit accuracy, solvability, and complexity; (2) the priority score calculation algorithm to select the input with the most critical branches. This solution can schedule inputs and branches in a sequence that will enhance the performance of hybrid fuzzing in the best way with acceptable overhead. Our approach is based on dynamic analysis by instrumentation in run-time, which is adaptive to different situations.
- We implement a hybrid fuzzer called SHFuzz, using our SHF approach, and compare it with the state-of-the-art hybrid fuzzer as a baseline. The evaluation shows that our solution can reduce the number of non-critical branches to solve and improve the performance of hybrid fuzzing on 20 real-world applications.

2. Background

In this section, we introduce the background of our work, which is divided into three major parts: coverage accuracy, input scheduling, and hybrid fuzzing.

2.1. Coverage Accuracy

For the coverage-based fuzzing method, the coverage is the main metric to evaluate the performance, though there are different ways to define how to compute coverage. The accuracy of different coverage computation methods is firstly discussed systematically by CollAFL [21], which also proposes a new method to improve this metric based on AFL [2], i.e., a widely-used fuzzer baseline. There are three different coverage computation methods: basic block coverage, edge or branch coverage, and path coverage. We use a simple program execution figure to explain their differences, as shown in Figure 1.

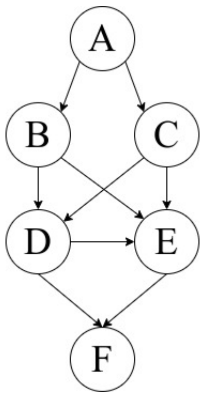


Figure 1. A simple program illustration with basic blocks and execution paths.

If we consider each basic block as a different coverage, then for the simple program shown in Figure 1, the full coverage of the program can be represented as set *{A, B, C, D, E, F}*. This metric is the least accurate one compared with the other two methods. For example, if we already have program paths *(A, B, E, F)* and *(A, B, D, F)*, the coverage set will be *{A, B, D, E, F}*. Then, if we discover a new execution path *(A, B, D, E, F)*, it will not be considered as a new path because all basic blocks in this path have already been recorded as explored coverage, and as a result, we will lose context.

Edge coverage describes more information about the execution path from one basic block to another, which considers the edge between two basic blocks as one unique coverage. If we use edge coverage to describe the program, it will be represented as set *{(A, B), (A, C), (B, D), (B, E), (C, E), (C, D), (D, F), (D, E), (E, F)}*. Though edge coverage is more accurate than basic block coverage, it is still not perfect. For example, suppose that we have already found program paths *(A, B, D, E, F)* and *(A, C, D, F)*, then our coverage set *P* will be *{(A, B), (B, D), (D, E), (E, F), (A, C), (C, D), (D, F)}*. If we then find another path *(A, C, D, E, F)*, which is a new path that has never been executed, however, it will not be considered under the edge coverage metric, because the edge coverage is set *Q {(A, C), (C, D), (D, E), (E, F)}*, which is the subset of the previously calculated set *P*. As a result, we will lose program coverage context.

This simple example shows clearly that path coverage is the most accurate way of computing coverage, edge coverage is less accurate, and basic block coverage is the least accurate. However, path coverage is hard to apply to real-world scenarios because of the huge overhead for computation. We need heavy instrumentation analysis on the program to get precise path coverage information. Therefore, edge coverage is a good trade-off in terms of both accuracy and scalability.

When we try to selectively execute branches in concolic execution, the problem of coverage accuracy still influences the results. Specifically, it is important to decide whether a branch is executed or not. If we only consider each branch as unique, then we will lose the context of edge coverage, similar to the problem we have shown in the previous paragraphs. To solve this problem, we propose a critical branch selection algorithm with a new way to calculate the branch coverage and decide whether a branch should be executed or not.

2.2. Input Scheduling

Input scheduling is an important part of fuzzing engines, no matter which kind of fuzzing strategy they use. Besides hybrid fuzzing, works are focusing on seed selection in both coverage-based fuzzing and directed fuzzing. In 2014, Alexandre Rebert et al. [22] firstly proposed a systematic analysis for the seed selection scheduling problem. They tested six different seed selection strategies on Peach Fuzzer [23], for example, by minimizing the seed input set according to different features like input size and execution time. The conclusion is that the seed selection algorithm can influence the performance of the fuzzer engine, and a well-designed strategy is better than random seed selection. Most fuzzing engines have their input scheduling strategies instead of random selection.

Early coverage-based fuzzing works [2,24,25] focused more on the inputs, which can introduce new coverage to the execution, however lacking knowledge on where these new inputs will lead. Thus, new works came out and considers the information on unique paths to define the quality of different inputs, such as [3,26]. In general, the intuition of choosing inputs is obvious: we should choose the inputs that can be mutated to reach unexplored branch paths.

CollAFL [21] introduces three different strategies to solve the input scheduling problem in a more fundamental way: (a) the memory-access guided strategy, which prefers inputs with more memory-access operations; (b) the untouched-neighbor-branch guided policy, which is based on static analysis to find unexplored paths; (c) the untouched-neighbor-descendant guided policy. These strategies do not apply to our goals. Firstly, our approach aims at increasing coverage, so that the memory-read directed strategy is of no use to our target. Secondly, in hybrid fuzzing, each branch will be negated to a certain unexplored path, so that it is not necessary to trace the information of neighbor branches. In hybrid fuzzing, a more important problem is branch selection, which we will explain later in the next subsection.

2.3. Hybrid Fuzzing

Hybrid fuzzing was firstly introduced in 2007 [15] and then brought to the public as a hot topic by Driller [9] in 2016. Both works used the demand launch strategy, which starts the fuzzing in the beginning, then when the fuzzing process gets stuck, concolic execution is triggered to analyze the selected inputs and negate branches to get new test cases, which can introduce new coverage. There are two disadvantages in this demand launch strategy, as described in DigFuzz [11]. Firstly, it is hard to define the state of "stuck". In the experiments of DigFuzz, for most of the testing programs, concolic execution is never triggered in the predefined test time because the fuzzer never becomes "stuck". Secondly, because of the bad efficiency of concolic execution, it takes too long to analyze even only one input, which makes the overall performance worse than pure fuzzing in some cases.

MDPC [19] proposes a new strategy called optimal switch, which uses the optimal decision on which branches to solve by concolic execution. More specifically, the MDPC algorithm calculates the complexity for every branch in terms of solving, then decides which branches will be solved by concolic execution and which branches will be solved by fuzzing. The problem of this work is that the optimal decision costs too much overhead since the analysis is performed on every branch. However, this work is still important because it firstly introduced the concept of branch scheduling to hybrid fuzzing.

DigFuzz [11] uses the same strategy as MDPC, which also calculates branch complexity, but with a smarter algorithm called MCP3. By considering fuzzing as a process of random sampling, this algorithm builds an execution tree with probabilities between different branches. For each branch on the execution tree, there is a probability to get the result of this branch by fuzzing. Thus, the probability of one path will be the product of all branches along this path. DigFuzz uses coverage statistics from AFL to build this execution tree for the whole program execution paths and schedule branches for fuzzing and concolic execution based on the probabilities calculated from the tree. For concolic execution, it will find the inputs that contain target branches and perform concolic execution on those inputs. There are a few problems that DigFuzz fails to solve. Firstly, it is still an optimized version of the demand launch strategy, which means concolic execution is not triggered as much as possible, at least not enough compared with the synchronization method. Secondly, it chooses the hardest-to-solve branches without considering unexplored coverage. Even though the selected branches are solved by concolic execution, there is a big chance that they may not contribute to the coverage or finding vulnerabilities. Thirdly, it fails to consider the solvability of the target branch, i.e., it fails to know if each branch has a solvable valid descendant path.

Our work is based on a more scalable and efficient synchronization strategy, which utilizes the power of multi-core CPUs. QSYM [10] is the state-of-the-art hybrid fuzzing work that provides an efficient concolic execution engine. Benefiting from the fast concolic execution engine, QSYM uses synchronization mode to collaborate with the AFL fuzzer by generating new test cases all the time in parallel with fuzzing. Even though this strategy can analyze much more inputs and branches than demand launch and optimal switch, input and branch scheduling is still very important. As we explained in the previous section, only a small amount of the candidate inputs will be analyzed in a 12 h hybrid fuzzing process. QSYM uses a naive approach to select inputs, by four metrics: (a) smaller inputs are favored; (b) the newer generated inputs are favored; (c) the inputs with new coverage are favored; (d) the original inputs are important. There is no coverage guidance on this input selection strategy, nor any branch selection method. In our experiment, we find that when analyzing a 218 byte png file on program libpng-1.2.56, QSYM needs more than 2 h to execute all the branches. However, if we focus only on the branches in the main binary, it needs only 2 min and 16 s, though the coverage decreases from 1005 source code lines to 827 lines. This shows that if we can choose the branches wisely, the efficiency of hybrid fuzzing will be enhanced greatly. Other hybrid fuzzing works also lack the analysis and design of input and branch scheduling. In our work, we try to analyze this problem systematically for the first time. To our best knowledge, our work is the first one to use a well-designed input and branch selection method in hybrid fuzzing with the synchronization strategy. In the next section, we will introduce our selective hybrid fuzzing approach, which is based on the synchronization mechanism.

3. Design

3.1. Overview

In this section, we propose the Selective Hybrid Fuzzing (SHF) approach, a new input and branch scheduling method based on the synchronization mechanism with the AFL fuzzer. As shown in Figure 2, we use the same synchronization framework as QSYM [10], the state-of-the-art work for hybrid fuzzing. For the fuzzer end, we create two instances, named master and slave, respectively. The QSYM concolic execution engine picks candidates from the test case queue of the slave fuzzer and then performs concolic execution. The newly generated test cases will be organized by the AFL naming standard and put into the QSYM directory, with the master and slave under the same category. The AFL fuzzer will then treat the qsym directory as another fuzzing instance and try to synchronize the newly generated inputs into the other two instances, i.e., master and slave. We add a new layer between the concolic execution engine and the fuzzers: a scheduler. The scheduler has three components: coverage calculation, branch selection, and input selection. The coverage calculation component uses binary instrumentation to calculate edge coverage based on basic blocks. The branch selection component then uses the coverage information and other information from dynamic binary analysis to determine the set of unexplored branches. The input selection component then chooses the inputs according to the selected branches and feeds them to the concolic execution engine in a specific sequence. With these three components as the middle-ware, we can make sure that every branch from a specific input can be useful to increase coverage instead of repeatedly re-exploring already discovered paths of the program.

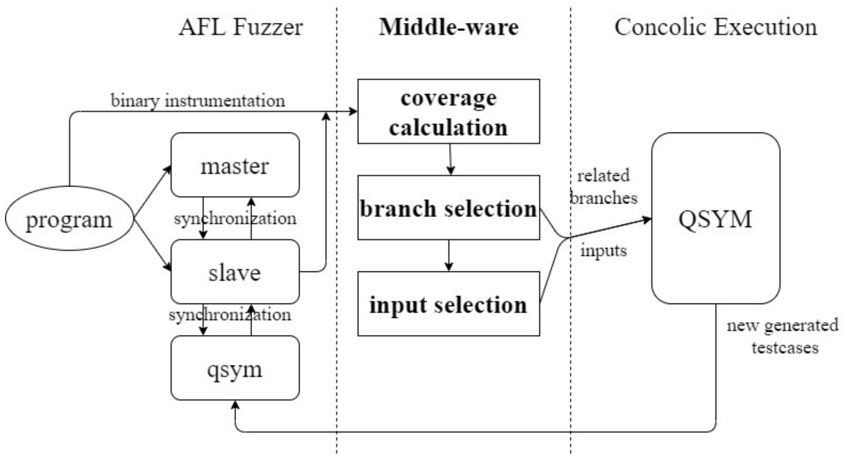


Figure 2. Overview of the Selective Hybrid Fuzzing (SHF) framework based on the AFL synchronization method.

The whole method of the SHF approach can be organized into three steps, as described below.

3.1.1. Step 1: Select Inputs from the Slave Fuzzer Queue for Further Analysis

In general, the SHF approach first chooses the critical branches and then selects the inputs with those branches to be analyzed by concolic execution. To calculate the critical scores of different branches, we need the information from running each input and analyzing each branch. Thus, the order of choosing inputs is important, because, for the branches chosen by an earlier analyzed input, they will not appear in the later analyzed ones. We take advantage of the input selection method used by QSYM, which is not mentioned in the paper, but in the source code [27]. The difference is that QSYM directly uses this method to select inputs for concolic execution, but we choose inputs in this order just to analyze the critical scores of different branches. We will then give a new execution order of inputs and related target branches by the information we get from further analysis.

In the first step, we choose inputs from the slave fuzzer queue in the order considering the following four facts:

- Inputs with new coverage are favored. If there is a "+cov" label in the inputs, it means that this input can introduce new coverage to the fuzzing process. We prioritize these inputs so that we can get more unexplored branches to analyze.
- The bigger inputs are favored. Normally, a bigger input means a longer execution path, therefore more new branches to explore. This is different from QSYM because it chooses inputs from the smallest size, for the reason that concolic execution on big inputs takes a very long time. We tested on one input and found out that the execution time increases dramatically as the input gets bigger, which is because there are too many branches to solve. Our approach only analyzes the critical branches instead of all of them, which mitigates this problem greatly. Thus, we can choose the largest sized inputs instead of the smaller sized ones to increase the length of the execution path.
- The newer generated inputs are favored. This fact is easy to understand because the newly generated inputs may have more information about the unexplored branches and code. In the AFL fuzzer queue, the newer generated inputs have bigger id numbers, which is easy to track.
- The original seed inputs are important. The seed inputs contain the most important information such as the file format. If we only apply the above three methods, then we will miss those original inputs. Thus, we prioritize the inputs with the "orig" label in the slave fuzzer queue.

At the end of this step, we will have an input list for further analysis, which is denoted as *input_list* in Algorithm 1.

3.1.2. Step 2: Calculate the Priority Score of Each Input and Related Target Branches by the Two Main Algorithms

In this step, we analyze each input in the order described in Step 1, calculate the critical score of each branch in one input by the Critical Branch Selection (CBS) algorithm, and select the target branches according to the score of each branch. Then, we give each input a priority score based on the target branches by the Priority Score Calculation (PSC) algorithm and output an input list that has order. The CBS and PSC algorithms are shown in Algorithm 1.

This algorithm takes the input list that was calculated in the first step as the input and gives an ordering input list as the output. There are two essential parts of this algorithm, which need further explanation. Firstly, we use three metrics to evaluate how critical a branch is: hit(H), solvability(S), and complexity(C). H means if one branch is explored or hit. S means the solvability, i.e., whether this branch can be solved or not. C means the complexity, i.e., how hard it is to solve this branch. We will explain this part in detail later. Secondly, it matters how the priority score is calculated for each input and how we will select inputs from the ordering input list. We will explain this part at the end of this section.

Algorithm 1 Selective hybrid fuzzing approach (CBS and PSC algorithms).

```
Input: input_list
Output: ordering_input_list
1:  for each input in input_list do
2:      branches = get_branches(input)
3:      for each branch in branches do
4:          H = if_already_hit(branch)
5:          S = get_solvability(branch)
6:          if H == 0 and S == 1 then
7:              target_branch_list.append(branch)
8:          end if
9:      end for
10:     for branch in target_branch_list do
11:         C[branch] = get_complexity(branch)
12:     end for
13:     sort C by inverted order
14:     final_branch_list = {select top 50% branches from C with most complexity}
15:     input_score = get_input_score(final_branch_list)
16:     priority_score_dict[input] = input_score
17: end for
18: ordering_input_list = sort priority_score_dict by score
19: return  ordering_input_list
```

3.1.3. Step 3: Select the Inputs and Related Branches for Concolic Execution

After the previous steps, we will have an ordering input list, from which we can select inputs by the order of the highest priority score. Besides, we also have the selected branches related to each input. Thus, we can pick up one input from the list by the pre-defined order and perform concolic execution on this input, then analyze the related selected branches. In theory, these branches with the inputs will have the best efficiency by only executing and analyzing necessary branches instead of all of them. In the evaluation section, we will show how our method performs in 20 different real-world applications.

3.2. Critical Branch Selection

Algorithm 1 shows that for one input, we calculate three values for each branch: *H* (hit), *S* (solvability), and *C* (complexity). Based on these three pieces of information, we calculate the priority score for each input. In this section, we will explain in detail what these three values mean and how they are calculated. The intuition of selecting critical branches is obvious: firstly, the branch should not be analyzed before; otherwise, it will waste time on the same coverage; secondly, the branch should be able to be solved; thirdly, we should give the concolic execution the most complex constraints and let the fuzzer solve less complicated constraints. Our CBS algorithm is based on these three assumptions.

3.2.1. Hit Accuracy

The first assumption shows that we should always focus on the "unexplored" branches, which are destined to introduce new coverage by generating new test cases. In theory, we should choose branches that have never been analyzed before, instead of repeatedly executing the already solved branches from time to time, which will not introduce any new coverage. However, in practice, it is hard to distinguish the "unexplored" branches from the already solved ones.

The problem originates from the coverage analysis, which we have explained in the previous section. For each branch with a particular address, we can not easily decide if this branch is already explored or not. We can see the problem clearly shown in Figure 3. Let us say if we have already solved Branch 3 once, by Input 1 guiding the execution path: *(A, B, D, E)*, then we will have a new execution coverage: *(A, B, D, F)*. Then, there is another Input 2 that can guide execution path: *(A, C, D, E)*. This input can introduce new coverage because a new basic block *C* is executed. Now, the problem happens: should we analyze Branch 3 in Input 2? If we believe that Branch 3 has already been solved once and do not analyze this branch any more, then we will lose the execution path *(A, C, D, F)*, which can be introduced by solving Branch 3 in Input 2. However, if we consider that Branch 3 under this situation is not explored, how can we distinguish it from the previously solved one in Input 1?

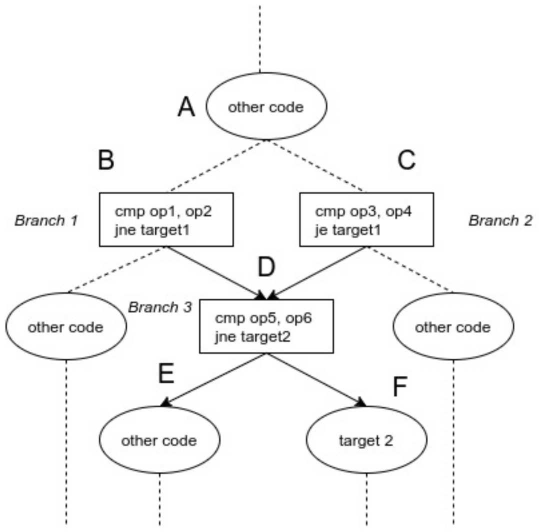


Figure 3. The problem of defining whether a branch is explored or not.

In the Background section, we introduce different ways of calculating coverage. If we want to utilize a more precise calculation of coverage, then we should consider Branch 3 in different execution paths as differently-explored branches. To solve this problem, we propose the "hit" metric to measure if one branch is explored or not, which uses a new block-edge coverage tracing algorithm by dynamic binary instrumentation. We define the tuple *(previous branch, target branch)* as a unique branch. For example, in Figure 3, Branch 3 in Input 1 will be denoted as *(Branch 1, Branch 3)*, while Branch 3 in Input 2 will be denoted as *(Branch 2, Branch 3)*. Thus, the two different Branch 3s will be distinguished well and both be analyzed by concolic execution. In this way, we can mitigate the imprecise problem of block coverage, which is caused by only considering the branch address as unique.

We use a dynamic instrumentation tool to run each input and record the execution trace of all branches. When analyzing branches in concolic execution, we define the previous branch as the hash value of the current branch and record the previous branch value in a hash dictionary with the current branch as key and a previous branch value list as the hash value. In the *if_already_hit()* function shown in Algorithm 1, when analyzing one branch, we will look up in the hash dictionary to see if this branch has a hash value in it. If not, then this branch is "not hit", and we will set the H value of this branch to zero. Otherwise, we will consider this branch as "hit" and set the H value to one.

3.2.2. Solvability

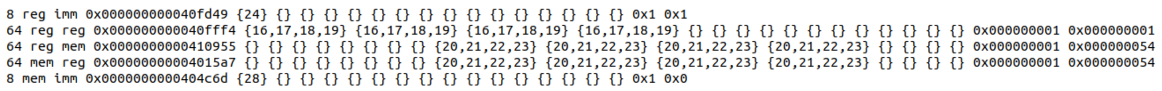
Not every branch has two successor targets that can be negated to get another execution path. If a branch cannot be solved by negating, we should not try to solve it from the beginning. We use static analysis to find out if one branch has two different successor basic blocks. Intel Pin [28] provides an analysis interface for this kind of static analysis, by which we can get the taken and non-taken basic blocks' information related to a particular branch. When running each input with our pintool, we record the taken and non-taken paths of each branch. In the *get_solvability()* function shown in Algorithm 1, for each branch, if it does not have a non-taken path, i.e., the value is zero, then we will consider this branch as "not solvable" and set the S value of solvability to zero. Otherwise, we will consider the branch as "solvable" and set the S value of solvability to one.

3.2.3. Complexity

Fuzzer runs very fast to generate inputs getting a shallow path coverage, while concolic execution is better at solving complicated branch constraints to reach deeper code in the program. In theory, we would like to make the fuzzer focus on the easy-to-mutate inputs that can get less complicated branches, while concolic execution focuses on solving the complicated branches. The question is how to define complexity.

We took the idea from the way fuzzers function. Normally, fuzzers mutate the selected input by different heuristics to get more coverage. The more input offsets a fuzzer needs to mutate to get to a certain point, the bigger the search space is; therefore, the harder this branch is to solve. For example, if a particular branch is related to five different offsets in the input, then the search space will be 2^{40} (five bytes, each byte with eight bits). The intuition is obvious: the more offsets are related to a branch, the harder the branch is to solve. If we can get the information on how many offsets will influence each branch, then we will get an approximate estimation of how hard the mutation is for each branch. That is to say, we should choose branches with the most related bytes in the inputs.

We use dynamic taint analysis to trace which offsets of the input can influence a branch. By running the taint analysis, we taint the input and then trace all compare instructions to see which offsets in the input will influence each compare instruction. After running the taint analysis tool, we can get results as shown in Figure 4. The first three columns show information about the operands of compare instructions. The fourth column shows the address of each compare instruction. The fifth to twentieth columns show which offsets of the input will influence each byte of operand. Therefore, the union set of these offsets will influence the result of this compare instruction.



```
8 reg imm 0x000000000040fd49 {24} {} {} {} {} {} {} {} {} {} {} {} {} {} {} {} 0x1 0x1
64 reg reg 0x000000000040fff4 {16,17,18,19} {16,17,18,19} {16,17,18,19} {16,17,18,19} {} {} {} {} {} {} {} {} {} {} {} {} 0x000000001 0x000000001
64 reg mem 0x0000000000410955 {} {} {} {} {} {} {} {} {20,21,22,23} {20,21,22,23} {20,21,22,23} {20,21,22,23} {} {} {} {} 0x000000001 0x000000054
64 mem reg 0x00000000004015a7 {} {} {} {} {} {} {} {} {20,21,22,23} {20,21,22,23} {20,21,22,23} {20,21,22,23} {} {} {} {} 0x000000001 0x000000054
8 mem imm 0x0000000000404c6d {28} {} {} {} {} {} {} {} {} {} {} {} {} {} {} {} 0x1 0x0
```

Figure 4. Example of the results by dynamic taint analysis.

Our method is to get all tainted bytes that will influence each compare instruction and calculate how many offsets there are related to each compare instruction. We consider the number of tainted offsets as the complexity value of each branch. In the *get_complexity()* function, we do taint analysis on the input and get the number of tainted offsets related to each branch, then assign this number as complexity value C for each branch. The output of this function is a dictionary of branches and related complexity values. We then sort this C dictionary by the inverted order, choose the top 50% of the branches, and return a final branch list. Therefore, we will have a branch list related to one input, containing the hardest to solve branches from this input.

3.3. Priority Score Calculation

The intuition of input scheduling is to choose the inputs that can generate the most new coverage by concolic execution. Therefore, we need more valid different target branches as possible. It would be best if all of these branches will guide to unexplored code. The priority score calculation algorithm is based on the input priority score, which is decided by the number of critical branches.

We use four steps to calculate the priority score related to each input, with the information generated by the branch scheduling part.

- Choose the branches with the H value as zero. Firstly, we consider the H value, which denotes if a branch is explored or not. We will choose these branches with the H value as zero for each input, which means they are not explored.
- Choose the branches with the S value as one. Secondly, we consider the S value, which denotes if a branch can be solved or not. We will choose these branches with the S value as one for each input, which means they can be solved.
- Calculate the complexity of the chosen branches. Thirdly, we consider the C value, which denotes the complexity to solve a branch. We choose the top 50% of branches with the biggest C values.
- Calculate the priority score for each input. Lastly, we calculate the priority score for each input by the number of critical branches chosen in the previous three steps, as Algorithm 1 shows.

With the priority score, we can have an ordering input list and choose inputs sequentially from this list, then run concolic execution on each input with related target branches. There is another thing worth mentioning, regarding the sequence that we follow to consider each value. We believe that it is more important to explore untouched paths than solve simple constraints. Therefore, we consider the hit and solvability value before the complexity value. Furthermore, the percentage of critical branches can always be changed for concrete situations. If there are not many branches in one run, we can choose more than half of the branches. We can also define the exact number of branches to choose.

4. Implementation

In this section, we introduce how we implement the selective hybrid fuzzing approach on a hybrid fuzzer named SHFuzz.

4.1. Scheduler Implementation

Hit and solvability calculation: We implement a pintool by Pin [29] binary instrumentation to record all executed basic blocks and calculate block-edge coverage related to each target branch, then we can get the value of H. We also implement the solvability calculation part by static analysis using the pintool interface to trace the taken and non-taken target of each branch so that we can get the value of S.

Complexity calculation: We use an existing taint analysis tool libdft64 provided by the VUzzer 64 bit version [30]. Libdft [31] is a dynamic taint analysis tool, which has only a 32 bit implementation as the original version and later is modified to apply to 64 bit systems. By this tool, we can get the tainted offsets in the input related to each branch, so that we can calculate the value of C based on this information.

Overall scheduler: We implement the scheduler by a Python script to analyze each input in a specified order and then use the above two calculation tools to get the essential information and calculate the final ordering input list. We observe that for each input, the calculation time is less than 2 s, then we can deal with 21,600 input files in 12 h, which is much better than the original input selection method in QSYM, which can only execute 480 inputs theoretically in 12 h. This scheduler will be running in parallel with AFL fuzzers and the QSYM concolic execution engine.

4.2. SHFuzz Modification

For the fuzzer end, we use AFL [2] and create two instances with the names of master and slave. For the concolic execution end, we use QSYM [10] as our evaluation baseline. In theory, the fuzzer and the concolic execution engine can be replaced by any existing tool, as long as some key functions can be implemented in the source code. For example, to add the scheduler to the entire framework, we need to modify the original QSYM source code to support input selection and branch selection, which means we have to make QSYM analyze only a specified list of branches.

Another thing worth mentioning is that our approach only focuses on binary applications and file-reading inputs. However, the principles of our method can be utilized on any application and hybrid fuzzing framework.

In general, we write 44 lines of C++ code to modify libdft64 to generate the information for further analysis. We write 416 lines of Python code to implement the main algorithm of input and branch scheduling to generate the target ordering input list. We write 139 lines of C++ code to modify QSYM to support branch selection. We write 73 lines of Python code for the integration of the whole hybrid fuzzing framework with our scheduler middle-ware.

5. Evaluation

In this section, we test our approach on 20 applications with various functions and program logic. We use the QSYM original implementation as a baseline and compare it with SHFuzz. The results show that after applying our SHF approach, hybrid fuzzing can generate more coverage and unique crashes to find more bugs in less time. We run each program for 12 h and record the numbers of coverage and unique crashes.

Environmental setup: We run all experiments on two identical machines with the Ubuntu 16.04 LTS operating system, Intel Xeon CPU E5-2630 (2.40GHz, 32 cores), and 128GB RAM.

5.1. Dataset

We evaluate 20 real-world applications with different functions and various program logic. Eighteen of them are from the Google Fuzzer Test Suite [20], which have known existing vulnerabilities and make evaluation more practical to compare the capability of finding bugs. The other two applications are the common utilities with the latest version. All 20 applications represent different aspects of functions and program logic, such as image file processing, server program, database processing, string processing, audio file processing, etc. The various functionalities can represent different kinds of constraints and branches that hybrid fuzzing will deal with, which is a good example set for evaluation. The types of different applications and other information are shown in Table 1.

Table 1. Types and other information of the 20 evaluated real-world programs.

Name	Version	Src	Size (KB)	Type
boringssl	2016-01-12	GFTS [20]	2244	server program
c-ares	CVE-2016-5180		14	asynchronous DNS resolves
guetzli	2017-3-30		3505	image processing
harfbuzz	1.3.2		3357	text type processing
json	2017-02-12		254	json file processing
libarchive	2017-01-04		1435	compression library
libjpeg-turbo	07-2017		663	image processing
libpng	1.2.56		429	image processing
libxml2	v2.9.2		4771	XML C parser
llvm-libcxxabi	2017-01-27		513	C++ compile tool
openssl	1.0.1f		2014	server program
openssl	1.0.2d		291	server program
pcre2	10.00		808	regular expression
proj4	2017-08-14		3395	cartographic projections library
re2	2014-12-09		4551	regular expression
sqlite	2016-11-14		895	database
vorbis	2017-12-11		320	audio file processing
woff2	2016-05-06		1725	font format
nm	2.34	binutils-2.34	4248	executable file information
readelf	2.34		1946	executable file information

5.2. Coverage Improvement

We run each application with QSYM and SHFuzz respectively under the same experimental setup for 12 h. Except for the input and branch scheduling part, the other experimental setups are the same. We set the time-out for each concolic execution process for one input as 90 s, the same as the default configuration of QSYM. Therefore, our evaluation can show fairly how our selective approach will improve the performance of hybrid fuzzing. The overall results of coverage and the number of crashes are shown in Table 2.

Table 2. Coverage, crash results, and bug found time of running QSYM and SHFuzz for 12 h on 20 applications.

	Method	Coverage	Crash	Bug Found Time
c-ares-CVE-2016-5180	QSYM	42	7	0:29:12
	SHFuzz	42	11	0:01:27
guetzli-2017-3-30	QSYM	3741	1	7:15:46
	SHFuzz	4300	18	3:16:16
json-2017-02-12	QSYM	3186	17	0:05:30
	SHFuzz	3149	24	0:01:42
llvm-libcxxabi-2017-01-27	QSYM	10,559	182	0:15:40
	SHFuzz	11,458	243	0:11:29
openssl-1.0.2d	QSYM	3276	35	0:54:50
	SHFuzz	3226	84	0:28:10
pcre2-10.00	QSYM	23,121	23	0:47:59
	SHFuzz	25,633	570	0:18:01
nm-2.34	QSYM	6969	22	8:39:31
	SHFuzz	6845	39	0:22:11
libxml2-v2.9.2	QSYM	5513	0	-
	SHFuzz	5833	6	0:18:31

Table 2. *Cont.*

	Method	Coverage	Crash	Bug Found Time
vorbis-2017-12-11	QSYM	3028	0	-
	SHFuzz	3360	194	2:16:45
boringssl-2016-01-12	QSYM	1529	0	-
	SHFuzz	1613	0	-
harfbuzz-1.3.2	QSYM	11,995	0	-
	SHFuzz	12,093	0	-
libarchive-2017-01-04	QSYM	5273	0	-
	SHFuzz	5836	0	-
libjpeg-turbo-07-2017	QSYM	4135	0	-
	SHFuzz	4265	0	-
proj4-2017-08-15	QSYM	1286	0	-
	SHFuzz	1336	0	-
re2-2014-12-09	QSYM	5305	0	-
	SHFuzz	5456	0	-
sqlite-2016-11-14	QSYM	7313	0	-
	SHFuzz	10,835	0	-
woff2-2016-05-06	QSYM	1208	0	-
	SHFuzz	1547	0	-
openssl-1.0.1f	QSYM	557	0	-
	SHFuzz	623	0	-
libpng-1.2.56	QSYM	1273	0	-
	SHFuzz	1365	0	-
readelf-2.34	QSYM	14,266	0	-
	SHFuzz	14,718	0	-

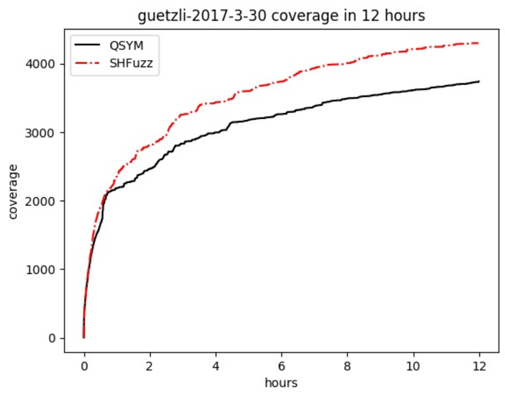
From Table 3, we can see the increasing number and percentage of the results by SHFuzz compared with the QSYM original configuration. As for the coverage part, we can see clearly that 16 out of 20 applications show an obvious increase in the number of coverage. Though the other four show negative results on coverage, they still can find more bugs compared with QSYM. From the positive results, we can see that sqlite-2016-11-14, pcre2-10.00, llvm-libcxxabi-2017-01-27, and guetzli-2017-3-30 show the best improvement by SHFuzz, each increasing the coverage by 3522 (48.16%), 2512 (10.86%), 899 (8.51%), and 559 (14.94%). We also calculate the average improvement on coverage, which is 497 (8.40%) more lines of coverage. These data also include negative results, so we believe they show that for most applications, our SHF approach can help greatly with increasing coverage compared with the original configuration in hybrid fuzzing.

We show the fuzzing process figures of four applications with the best improvement on coverage in Figure 5, which shows why SHFuzz works. In each sub-figure, the X-axis shows the time range, while the Y-axis shows the coverage hybrid fuzzing has reached at each hour. We can see that for the QSYM method, it is hard to increase the coverage after some time, where the curve tends to be flat. On the contrary, SHFuzz still shows potential increase trends. This is because at the same time-out, since QSYM is trying to calculate all branches without any selection, it will always be stuck in the same shallow branches and cannot go deeper into the program paths. However, SHFuzz can always try to solve the most critical branches, which can guide the execution to deeper paths.

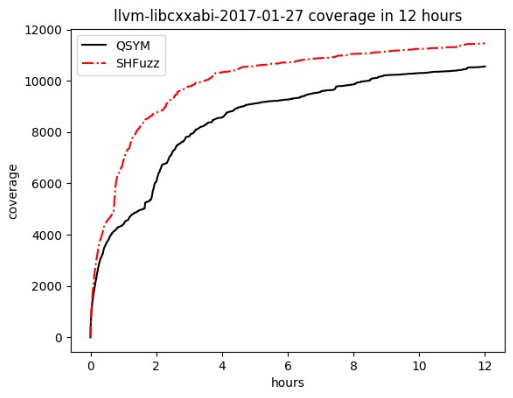
Table 3. Increased number of coverage, crashes, and bugs found time by SHFuzz compared with QSYM.

	Coverage	Percentage	Crash	Bug Found Time
c-ares-CVE-2016-5180	+0	+0%	+4	20.13×
guetzli-2017-3-30	+559	+14.94%	+17	2.22×
json-2017-02-12	−37	−1.16%	+7	3.23×
llvm-libcxxabi-2017-01-27	+899	+8.51%	+61	1.36×
openssl-1.0.2d	-50	−1.57%	+49	1.96×
pcre2-10.00	+2512	+10.86%	+547	2.66×
nm-2.34	-124	−1.78%	+17	23.42×
libxml2-v2.9.2	+320	+5.80%	+6	new
vorbis-2017-12-11	+332	+10.96%	+194	new
boringssl-2016-01-12	84	+5.49%	0	-
harfbuzz-1.3.2	+98	+0.82%	0	-
libarchive-2017-01-04	+563	+10.67%	0	-
libjpeg-turbo-07-2017	+130	+3.14%	0	-
proj4-2017-08-15	+50	+3.88%	0	-
re2-2014-12-09	+151	+2.84%	0	-
sqlite-2016-11-14	+3522	+48.16%	0	-
woff2-2016-05-06	+339	+28.06%	0	-
openssl-1.0.1f	+66	+11.85%	0	-
libpng-1.2.56	+92	+7.22%	0	-
readelf-2.34	+452	+3.17%	0	-
average	+497	+8.40%	+100	7.85×

For the negative results in nm-2.34, openssl-1.0.2d, and json-2017-02-12, each decreases coverage by 124, 50, and 37, which are rather small numbers. We can see that SHFuzz still generates more unique crashes and can find the existing bugs faster than QSYM in these three applications, showing the strength of SHFuzz in finding deeply-hidden bugs. We will explain this in detail in the next subsection.

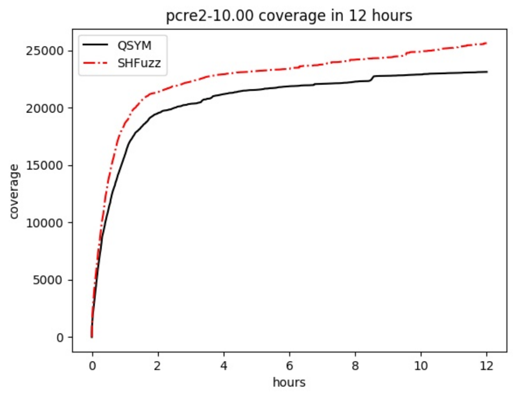


(**a**) guetzli-2017-3-30 results by QSYM and SHFuzz.

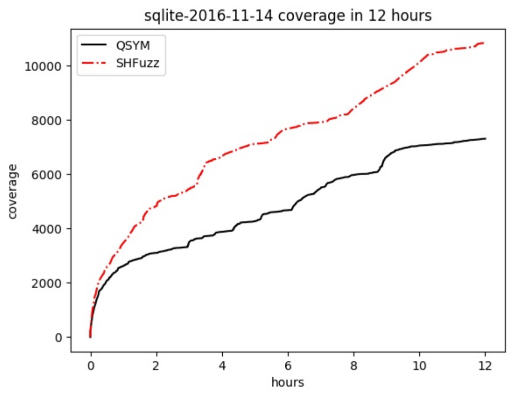


(**b**) llvm-libcxxabi-2017-01-27 results by QSYM and SHFuzz.

Figure 5. *Cont.*



(**c**) pcre2-10.00 results by QSYM and SHFuzz.



(**d**) sqlite-2016-11-14 results by QSYM and SHFuzz.

Figure 5. Coverage improvement by SHFuzz compared with QSYM shown by four applications.

5.3. Find More Bugs

The ultimate goal of fuzzing is to find more bugs. Our evaluation results show that with the help of our SHF approach, hybrid fuzzing can find more unique crashes compared with the original configuration, as shown in Table 3. On average, SHFuzz can find 100 more new crashes than the QSYM original configuration in each of the 20 applications, which improves the original implementation. It is worth mentioning that for the three applications with negative results on coverage, all of them can find more bugs compared with QSYM. More specifically, nm-2.34, openssl-1.0.2d, and json-2017-02-12 can generate 17, 49, and 7 more unique crashes, respectively.

For the 20 applications, eighteen of them are from Google Fuzzer Test Suite, which have existing known bugs in the program. We analyze the crashes generated by QSYM and SHFuzz, respectively, and calculate the earliest found time of existing bugs in the program, as shown in Table 2. It shows clearly that for all seven applications where QSYM and SHFuzz can both find crashes, SHFuzz can find the existing bugs much earlier than QSYM. Besides, SHFuzz can find bugs in two applications where QSYM fails to find them. The two applications with negative results on coverage, i.e., openssl-1.0.2d and json-2017-02-12, also show effectiveness on finding existing vulnerabilities in a shorter time by SHFuzz compared with QSYM. On guetzli-2017-3-30 and nm-2.34, SHFuzz can reduce the found time of crashes as much as hours, while on libxml2-v2.9.2 and vorbis-2017-12-11, SHFuzz can find the vulnerability that QSYM fails to find. On average, SHFuzz can find the existing bugs 7.85× faster than QSYM. These results show that our SHF approach can help find more bugs in less time compared with existing state-of-the-art hybrid fuzzing work.

For another application, nm-2.34, which comes from the latest version of binutils, we still find 39 new crashes by SHFuzz, 17 more than QSYM. We analyzed the crashes and they turned out to be an already-solved issue [32], which was found by other groups recently. This result shows that our selective hybrid fuzzing approach works at finding new bugs more efficiently than existing works.

5.4. Discussion

In the evaluation, we show that our selective hybrid fuzzing approach can reduce the number of non-critical branches for concolic execution to solve and improve coverage performance and the bug finding ability in hybrid fuzzing. However, there are still some unsolved problems related to this topic. Firstly, we analyzed the solvability by static analysis, which may have a false-positive problem, because of implicit data flow. A possible solution is to analyze the solvability by dynamic analysis, which may introduce extra overhead. Secondly, we use taint analysis to get the complexity information, which could be hindered by the over-tainting and under-tainting problems. This is the limitation of the taint analysis tool. A possible solution is to implement it with another taint analysis tool that has better precision; however, this will also introduce more overhead. Thirdly, our block-edge coverage calculation method is more precise than basic block coverage; however, it is still a coarse way to calculate coverage. The best solution would be to use path coverage or add more edges to calculate the coverage. In general, we trade accuracy for efficiency. Despite the unsolved problems, our work is the first attempt to solve the branch selection problem in synchronization-based hybrid fuzzing. We will leave these problems to further study.

Another thing to mention is the overhead of the SHFuzz algorithm. The main extra overhead is caused by running dynamic taint analysis. In our observation, for all 20 applications, the average time cost for one input is less than 2 s. Since we only execute taint analysis for each input once, the overall time overhead cost by the SHFuzz algorithm will approximately be 30 min (considering analyzing 1000 inputs for each application), which is acceptable for the evaluation time of 12 h.

6. Related Works

6.1. Code Coverage Accuracy

The definition of coverage accuracy will influence the efficiency of different fuzzers and also the way they try to mutate the inputs. VUzzer [3], libFuzzer [24], and honggfuzz [25] use basic block coverage. VUzzer uses Pin [29] to instrument binaries and get essential basic block information for mutation. LibFuzzer and honggfuzz use SanitizerCoverage [33] to compute basic block coverage. AFL [2] uses edge coverage, which is more accurate than the previously mentioned work; however, it has very bad hash functions, which cause collision problems. CollAFL [21] proposes a new hash function to compute edge coverage and mitigate the collision problem successfully.

In our paper, we learned the lessons from the failure of AFL and other previous works. We propose the critical branch selection algorithm partially based on the hit accuracy of each branch, to get the information of whether a branch has been executed or not. To define the hit accuracy, we consider the tuple of *(previous branch, target branch)* as a unique branch. In other words, for a target branch, if it is already solved with the same previous branch, then we will consider this branch as "already hit". Since the number with different branches is far less than the number of unique edges, it is acceptable to record the branch tuple without applying any hash algorithms. Thus, we avoid the problem of hash collision, as well as maintain a low overhead of recording useful information.

6.2. Input Scheduling

Coverage-based fuzzing: Honggfuzz [25] sequentially chooses the input to mutate from the candidate list, using the first-in, first-out strategy. LibFuzzer [24] prefers the inputs that can hit more new basic blocks, using SanitizerCoverage [33] to trace the information of basic block coverage. AFL [2] uses edge coverage information to choose the seed input, based on which seeds can find new edge coverage. Besides, it also prefers cases with a smaller size and is executed faster, to execute as many files as possible in a fixed time. AFLFast [26] observes that in the fuzzing process, most resources are wasted on repeatedly executing the same high-frequency paths, while low-frequency paths are rarely executed. To solve this problem, they used the Markov chain to discover low-frequency paths and prioritize the inputs that include those low-frequency paths. This work is based on AFL [2], using the same edge coverage hash algorithm as AFL to trace the frequency of paths. VUzzer [3] has a similar strategy, which is based on basic block coverage. This work defines the "hard-to-reach" paths by calculating the weights of different basic blocks in terms of executing probabilities. The error-handling basic blocks are excluded from the weight so that the inputs with these basic blocks will be considered as non-essential. If a basic block is executed too many times, then the weight of this basic block will also decrease. In summary, VUzzer chooses inputs with deeper paths to be executed and mutated by the fuzzer. Angora [5] uses this method by source code instrumentation to get the information of unexplored branches. In our work, we use a similar strategy, however on binary instrumentation, which is more challenging to achieve.

Directed fuzzing: Normally, directed fuzzing is based on target locations that are discovered by static analysis or other information about the vulnerabilities. For input scheduling, most works choose to find inputs that are more likely to reach the target locations. AFLGo [34] selects seed inputs that are closer to the target location. QTEP [35] selects inputs that can discover more faulty code by statically analyzing the program, to increase the probability of finding new vulnerabilities. Both works rely on static analysis, which has a high false-positive problem. SAVIOR [36] is a bug-driven approach that prioritizes the concolic execution of the seeds that are likely to uncover more vulnerabilities by modeling faulty situations using SMT constraints. In our paper, we only use binary analysis to analyze the priority of different inputs. Furthermore, we focus on the coverage more than specific target locations. Therefore, our approach to input scheduling is closer to most coverage-based fuzzing works, which treat unexplored branches as their most important concerns.

6.3. Hybrid Fuzzing and Branch Scheduling

Many hybrid fuzzing works focus more on increasing the scalability of concolic execution instead of scheduling inputs and branches, such as QSYM [10], Driller [9], and SynFuzz [37]. MDPC [19] and DigFuzz [11] use different strategies to calculate the importance of each branch and execute the branches in sequence. However, they are based on demand launch and optimal switch, respectively, which do not utilize the benefits of synchronization. QSYM [10] is the state-of-the-art hybrid fuzzing work that provides an efficient concolic execution engine, benefiting from the synchronization method. It uses four metrics to select inputs, but without any branch selection method. To our best knowledge, our work is the first one to use a well-designed input and branch selection method in hybrid fuzzing with the synchronization strategy.

Other existing works analyze the difficulty of a branch and the related new path; however, nearly all of them rely on very heavy analysis. Xie et al. [38] proposed value analysis to calculate the complexity of a path. Deepfuzz [39] uses probabilistic symbolic execution to find deeply hidden vulnerabilities. Those two works have high overhead on analysis, i.e., cost computational resources to analyze the program. If we need to put these many resources toward estimating the complexity of each branch, we could just perform concolic execution and solve the branches directly. In our work, we propose a lightweight method to estimate the importance of the branches in different inputs by considering coverage (hit accuracy), complexity, and solvability, which does not cost much overhead compared with the original configuration.

7. Conclusions

In this paper, we propose the selective hybrid fuzzing approach with two main algorithms: critical branch selection and priority score calculation. The CBS algorithm calculates the critical score of each branch by three metrics, hit accuracy, solvability, and complexity, and then selects the critical branches according to the scores. The PSC algorithm selects inputs with the most critical branches. In the evaluation, we show that our SHF approach can achieve 8.40% more coverage and 100 more unique crashes in 20 applications on average and can find the existing bugs 7.85× faster than the state-of-the-art work. Our implementation is based on binary instrumentation, which does not need source code. To our best knowledge, our work is the first to systematically analyze the branch scheduling problem in hybrid fuzzing with the synchronization mechanism. We also discuss the limitations of our work and leave those unsolved problems to further study.

Author Contributions: X.M. contributed to the conceptualization, methodology, software, and writing the original draft of this paper. B.W. contributed to project administration. Y.T. contributed to funding acquisition. B.Y. contributed to the resources of the project. P.W. contributed to writing, including review and editing. All authors read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Key Research and Development Program of China (No. 2018YFB0204301), the Natural Science Foundation of Hunan Province in China (2019JJ50729), the Natural Science Foundation of China (61902412, 61902416), and the National University of Defense Technology Research Project (ZK20-17).

Acknowledgments: The authors would like to sincerely thank all the reviewers for their time and expertise for this paper. Your insightful comments helped us improve this work.

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

References

1. Klees, G.; Ruef, A.; Cooper, B.; Wei, S.; Hicks, M. Evaluating Fuzz Testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2123–2138.
2. Michal, Z. American Fuzzy Lop (AFL). Available online: http://lcamtuf.coredump.cx/afl/ (accessed on 20 May 2020).
3. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-Aware Evolutionary Fuzzing. In Proceedings of the NDSS Symposium 2017, San Diego, CA, USA, 26 February–1 March 2017.
4. Aschermann, C.; Schumilo, S.; Blazytko, T,; Gawlik, R.; Holz, T. REDQUEEN: Fuzzing with Input-to-State Correspondence. In Proceedings of the 26th Annual Network and Distributed System Security Symposium, San Francisco, CA, USA, 24–27 February 2019.
5. 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.
6. Godefroid, P.; Kiezun, A.; Levin, M.Y. Grammar-based whitebox fuzzing. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008; pp. 206–215.
7. Blazytko, T.; Bishop, M.; Aschermann, C.; Cappos, J.; Schlogel, M.; Korshun, N.; Abbasi, A.; Schweighauser, M.; Schinzel, S.; Schumilo, S. GRIMOIRE: Synthesizing Structure while Fuzzing. In Proceedings of the USENIX Security, Santa Clara, CA, USA, 14–16 August 2019; pp. 1985–2002.
8. Gopinath, R.; Mathis, B.; Hoschele, M.; Kampmann, A.; Zeller, A. Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing. *arXiv* **2018**, arXiv:1810.08289.
9. 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.
10. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In Proceedings of the USENIX Security, Baltimore, MD, USA, 15–17 August 2018; pp. 745–761
11. 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.
12. Chipounov, V.; Kuznetsov, V.; Candea, G. The S2E Platform: Design, Implementation, and Applications. *Acm Trans. Comput. Syst.* **2012**, *30*, 1–49. [CrossRef]
13. Saudel, F.; Salwan, J. Triton: A Dynamic Symbolic Execution Framework. In Proceedings of the Symposium sur la sécurité des Technologies de l'Information et Des Communications, SSTIC, Rennes, France, 3–5 June 2015; pp. 31–54.
14. Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; John, G.; Feng, S.; Hauser, C.; Kruegel, C.; et al. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In Proceedings of the IEEE Symposium on Security and Privacy(SP), San Jose, CA, USA, 22–26 May 2016.
15. Majumdar, R.; Sen, K. Hybrid Concolic Testing. In Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, 20–26 May 2007; pp. 416–426.
16. Sen, K. Concolic testing. In Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering, Atlanta, GA, USA, 5–9 November 2007; pp. 571–572.
17. Baldoni, R.; Coppa, E.; D'elia, D.C.; Demetrescu, C.; Finocchi, I. A Survey of Symbolic Execution Techniques. *Acm Comput. Surv.* **2016**, *51*, 1–39. [CrossRef]
18. De Moura, L.; Bjorner, 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; pp. 337–340.
19. Wang, X.; Sun, J.; Chen, Z.; Zhang, P.; Wang, J.; Lin, Y. Towards optimal concolic testing. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 291–302.

20. Google Fuzzer Test Suite. Available online: https://github.com/google/fuzzer-test-suite/ (accessed on 20 May 2020).
21. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. CollAFL: Path Sensitive Fuzzing. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 679–696.
22. Rebert, A.; Cha, S.K.; Avgerinos, T.; Foote, J.; Warren, D.; Grieco, G.; Brumley, D. Optimizing seed selection for fuzzing. In Proceedings of the 23rd USENIX Security, San Diego, CA, USA, 20–22 August 2014; pp. 861–875.
23. Eddington, M. Peach Fuzzer. 2008. Available online: http://www.peachfuzzer.com/ (accessed on 20 May 2020).
24. Serebryany, K. Continuous fuzzing with libfuzzer and addresssanitizer. In Proceedings of the IEEE Cybersecurity Development (SecDev), Boston, MA, USA, 3–4 November 2016; p. 157.
25. Swiecki, R. Honggfuzz. 2016. Available online: http://code.google.com/p/honggfuzz (accessed on 20 May 2020).
26. Bohme, M.; Pham, V.; Roychoudhury, A. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Trans. Softw. Eng.* **2017**, *45*, 489–506. [CrossRef]
27. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. Available online: https://github.com/sslab-gatech/qsym (accessed on 20 May 2020).
28. Pin: A Dynamic Binary Instrumentation Tool. Available online: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool (accessed on 20 May 2020).
29. Luk, C.-K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; Wallace, S.; Reddi, V.J.; Kim, H. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Acm Sigplan Not.* **2005**, *40*, 190–200. [CrossRef]
30. VUzzer(64) Version 1.0. Available online: https://github.com/vusec/vuzzer64 (accessed on 20 May 2020).
31. Kemerlis, V.P.; Portokalidis, G.; Jee, K.; Keromytis, A.D. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, London, UK, 3–4 March 2012; pp. 121–132.
32. Bug 25447—Objcopy: Free() Invalid Pointer in _bfd_coff_free_symbols. Available online: https://sourceware.org/bugzilla/show_bug.cgi?id=25447 (accessed on 20 May 2020)
33. Sanitizercoverage: Clang documentation. Available online: https://clang.llvm.org/docs/SanitizerCoverage.html (accessed on 20 May 2020).
34. Bohme, M.; Pham, V.; Nguyen, M.; Roychoudhury, A. Directed Greybox Fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 2329–2344.
35. Wang, S.; Nam, J.; Tan, L. QTEP: Quality-aware test case prioritization. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 523–534.
36. 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 41st IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 18–21 May 2020.
37. Han, W.; Rahman, M.L.; Chen, Y.; Song, C.; Lee, B.; Shin, I. SynFuzz: Efficient Concolic Execution via Branch Condition Synthesis. *arXiv* **2019**, arXiv:1905.09532.
38. Xie, T.; Tillmann, N.; De Halleux, J.; Schulte, W. Fitness-guided path exploration in dynamic symbolic execution. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks, Lisbon, Portugal, 29 June–2 July 2009; pp. 359–368.
39. Bottinger, K.; Eckert, C. DeepFuzz: Triggering Vulnerabilities Deeply Hidden in Binaries. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, San Sebastián, Spain, 7–8 July 2016; pp. 25–34.