electronics

MDPI

Article

BSFuzz: Branch-State Guided Hybrid Fuzzing

Qi Hu [1,†], Weijia Chen [1,†], Zhi Wang [1], Shuaibing Lu [2,*], Yuanping Nie [2], Xiang Li [2] and Xiaohui Kuang [2]

[1] College of Cyber Science, Nankai University, Tianjin 300350, China; huqi@mail.nankai.edu.cn (Q.H.); weijiachen@mail.nankai.edu.cn (W.C.); zwang@nankai.edu.cn (Z.W.)
[2] National Key Laboratory of Science and Technology on Information System Security, Beijing 100085, China; yuanpingnie@nudt.edu.cn (Y.N.); ideal_work@163.com (X.L.); xiaohui-kuang@163.com (X.K.)
* Correspondence: lushuaibing@alumni.sjtu.edu.cn
† These authors contributed equally to this work.

Abstract: Hybrid fuzzing is an automated software testing approach that synchronizes test cases between the fuzzer and the concolic executor to improve performance. The concolic executor solves path constraints to direct the fuzzer to explore the uncovered path. Despite many performance optimizations for hybrid fuzzing, we observe that the concolic executor often repeatedly performs constraint solving on branches with unsolvable constraints and branches covered by multiple test cases. This can cause significant computational redundancies. To be efficient, we propose BSFuzz, which keeps tracking the coverage state and solving state in a lightweight branch state map. BSFuzz synchronizes the current coverage state of all test cases from the fuzzer's queue with the concolic executor in a timely manner to reduce constraint solving for high-frequency branches. It also records the branch-solving state during the concolic execution to reduce repeated solving of unsolvable branches. Guided by the coverage state and historical solving state, BSFuzz can efficiently discover and solve more branches. The experimental results with real-world programs demonstrate that BSFuzz can effectively increase the speed of the concolic executor and improve branch coverage.

Keywords: hybrid fuzzing; concolic execution; branch-solving state; coverage state

Citation: Hu, Q.; Chen, W.; Wang, Z.; Lu, S.; Nie, Y.; Li, X.; Kuang, X. BSFuzz: Branch-State Guided Hybrid Fuzzing. *Electronics* **2023**, *12*, 4033. https://doi.org/10.3390/electronics12194033

Academic Editor: Paulo Ferreira

Received: 2 September 2023
Revised: 21 September 2023
Accepted: 22 September 2023
Published: 25 September 2023

1. Introduction

Fuzzing is a widely used software testing method that quickly finds vulnerabilities in programs by randomly mutating seed inputs. However, due to its random nature, fuzzing can struggle to satisfy complex and tight constraints and may spend much time exploring repeated paths. To address this limitation, researchers have proposed hybrid fuzzing [1–5], which combines the strengths of fuzzing and concolic execution [6]. Fuzzing relies on random mutation to efficiently explore code, while concolic execution employs constraint solving to navigate complex and tight branch conditions. In a typical hybrid fuzzing framework, the concolic executor executes all test cases generated by the fuzzer, traces their execution, solves constraints at each branch, and generates test cases to achieve new code coverage. These new test cases are synchronized with the fuzzer, which starts a new round of mutations. This process continues iteratively, allowing the hybrid fuzzer to explore more code in a shorter amount of time.

Extremely expensive constraint solving is a challenge for hybrid fuzzing. Despite optimization efforts, hybrid fuzzers still face performance issues caused by the considerable computational overhead of symbolic emulation and constraint solving. The state-of-the-art hybrid fuzzers selectively solve only interesting path constraints to improve performance. Driller [1] invokes a concolic executor to help the fuzzer explore hard-to-pass branches when it gets stuck; i.e., when fuzzing can no longer increase code coverage. QSYM [2] implements instruction-level concolic execution via dynamic binary translation (DBT) and emulates only the instructions necessary to generate symbolic constraints, significantly reducing the number of symbolic emulations and improving the efficiency of concolic

execution. DigFuzz [3] and MEUZZ [7] use the Monte Carlo method and machine learning, respectively, to optimize the seed scheduling strategy of the concolic executor.

For test case generation, random mutation is much faster than constraint solving, which causes a delay in coverage state synchronization between the fuzzer and the concolic executor. For example, when the concolic executor executes a test case, the fuzzer may have already generated multiple new test cases. The delay in the coverage state synchronization may affect the judgment of when to invoke constraint solving. Additionally, we observe that the concolic executor constantly and repeatedly attempts to solve some constraints that cannot be solved. The concolic executor does not record the state of solving and repeatedly performs constraint solving when it encounters the same branch again. This can result in significant computational overhead and slow down the overall testing process. Finally, we find that the concolic executor does not distinguish between high-frequency and low-frequency branches when performing constraint solving. A high-frequency branch is one that has been covered by multiple test cases. As a rule of thumb, there are more uncovered branches near low-frequency branches. Therefore, it would be more efficient to spend more time exploring low-frequency branches rather than treating high-frequency and low-frequency branches equally.

In this study, we implemented a prototype system called BSFuzz, which integrates the coverage state of the fuzzer with the historical solving states of the concolic executor to improve constraint solving performance. We used a bitmap to record the number of times each branch is covered by test cases, as well as the constraint-solving state. When the concolic executor reaches a branch location, BSFuzz first determines whether the location can be successfully solved based on the historical solving states and skips locations that have been labeled unsolvable. Based on this strategy, we implemented BSFuzz-uns. In a second strategy, the concolic executor checks the execution frequency of branches and skips constraint solving for the high-frequency branches. We implemented BSFuzz-fre based on this strategy. In this way, BSFuzz can focus on solving the solvable and low-frequency branches to explore as much code as possible in a short time.

We tested BSFuzz and its two separate strategies, BSFuzz-uns and BSFuzz-fre, with six popular benchmark programs. Our evaluation results show that BSFuzz and its optimized strategies outperformed the current state-of-the-art hybrid fuzzer QSYM [2] in terms of branch coverage with six real-world programs. Our strategy effectively reduces the amount of redundant constraint solving and, on average, solves nearly 50% more branches. After 24 h of testing, the average bitmap coverage increased by 3.57% for BSFuzz, 2.32% for BSFuzz-uns, and 1.35% for BSFuzz-fre. These results demonstrate the effectiveness of BSFuzz and its optimized strategies in improving the efficiency of hybrid fuzzing.

This paper makes the following contributions:

- We analyzed mainstream hybrid fuzzers' synchronization and scheduling mechanisms. Our analysis revealed shortcomings of hybrid fuzzing in synchronizing test cases and selecting interesting branches for exploration;
- We designed and implemented a fast synchronization mechanism to synchronize the latest coverage state from the fuzzer to the concolic executor in real time and reduce solving computation for high-frequency branches;
- We propose the strategy of filtering out unsolvable branches. This strategy can be applied to all hybrid fuzzers to avoid meaningless branch solving;
- We evaluated our method's performance with real-world programs. Our experimental results demonstrated that BSFuzz outperformed our benchmark tool QSYM [2] in terms of branch coverage growth rate and efficiency.

2. Motivation

Hybrid fuzzing combines fuzzing and concolic execution based on test case synchronization and scheduling mechanisms. Test case synchronization refers to how the concolic executor synchronizes information from all test cases in the fuzzer, while scheduling refers to which branches the concolic executor performs constraint solving on. Based on our anal-

ysis, we found that current hybrid fuzzers suffer from drawbacks in both synchronization and scheduling.

2.1. Slow Concolic Execution

A significant number of test cases are generated during the fuzzing process. Typically, the concolic executor synchronizes all test cases generated by the fuzzer and performs constraint collection and solving. For example, both Driller [1] and QSYM [2] traverse and explore all inputs from the fuzzer. However, concolic execution must perform symbolic emulation and constraint solving, both of which have computational overheads, making concolic execution extremely slow. As a result, only a very small fraction of the test cases in the fuzzer queue can be explored by the concolic executor.

We conducted tests with a set of real-world benchmark programs using QSYM [2]. Table 1 shows that, after 24 h of execution, QSYM [2] only processed an average of 13.55% of the test cases in the fuzzer's queue. This indicates that the concolic executor was not able to keep up with the rate at which the fuzzer generated test cases. In all five programs except pngfix, the number of test cases processed by QSYM [2] was less than 10% of the total number of test cases generated by the fuzzer.

Table 1. Comparison between the numbers of test cases executed by QSYM [2] and generated by AFL [8] in 24 h.

Program	**AFL** [8]	**QSYM** [2]	**Ratio**
readelf	13,132	1177	8.96%
tcpdump	21751	1165	5.36%
pdfimages	15,521	880	5.67%
pdftops	16,875	865	5.13%
pdftotext	18,089	888	4.91%
pngfix	3194	1912	59.86%

There is a time gap between the concolic executor and the fuzzer. The concolic executor cannot process test cases as fast as the fuzzer generates them. This means that when the concolic executor generates a useful input for a particular path after a long period, this path may have already been covered by the newly generated input from the fuzzer. In this case, the input generated by the concolic executor is meaningless. Therefore, it is important for the concolic executor to obtain the latest coverage states of all test cases. The latest state can act as a good guide for the concolic executor, informing it when constraint solving should be performed and when it is not necessary. The concolic executor can thus focus its efforts on generating inputs for paths that have not yet been covered by the fuzzer's inputs.

2.2. Unsolvable Constraints

Existing hybrid fuzzers do not record whether the constraints are solved successfully when performing constraint solving. For example, when QSYM [2] encounters a branch position, it first tries to solve the complete path constraints. If the entirety of the constraints are *unsolvable*, QSYM [2] attempts to solve the last constraint of the path. Driller [1] only checks if the current branch can jump to an area of code not covered by fuzzing, without considering whether the fuzzer has attempt to solve the current branch multiple times but failed. The concolic executor continually solves the constraints of some branch positions that are not actually solvable, wasting time and resources.

We divide the constraint-solving states of one branch into three types: *solvable*, *partially solvable*, and *unsolvable*. If the full path constraints for the branch can be satisfied, it is *solvable*. If the full path constraints cannot be satisfied but the last constraint of the path can be solved, it is *partially solvable*. If even the last constraint of the path cannot be solved in the current context, it is *unsolvable*. The final constraint of a path is typically associated with a single judgment condition, making it less complex and easier to solve than the

constraints of the entire path. If the final constraint cannot be solved, then the constraints of the complete path cannot be solved either.

We found that the constraint-solving process involves many repeated attempts to solve unsolvable branches. We recorded the constraint solving information for pdftotext provided by QSYM's [2] concolic executor and present the results in Figure 1. The left figure shows the solving times over 24 h, where "Total" is the total solving time and "Unsolvable" is the solving time for the *unsolvable* branches. The right figure shows the number of branches solved in 24 h, where "Total" is the total number of branches, "Solvable" is the number of branches solved successfully, "Unsolvable" is the number of branches with *unsolvable* states, and "Unsolvable + solvable" is the number of branches with both *unsolvable* and *solvable* states demonstrated by repeated solving attempts in different contexts. The branches that could be successfully solved amounted to 27% of the total branches, and only 7% of all unsolvable branches were successfully solved after multiple iterations in different contexts. These branches amounted to 3% of the total branches. The solver consumed 44% of the total solving time on the unsolvable branches; i.e., the solver spent almost half of the time repeatedly attempted to solve the unsolvable branches but did not generate test cases that increased coverage. Therefore, we would like to reduce the solving time used for unsolvable branches and use the time spent on unsolvable branches for branches that may have solutions.

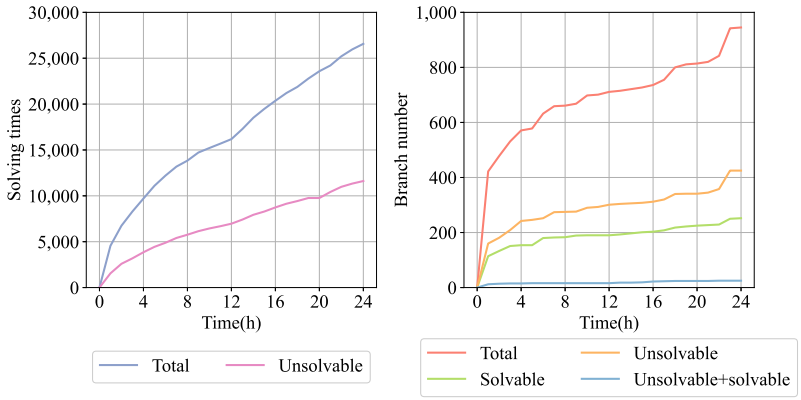

Figure 1. Example of branch solving for pdftotext.

2.3. High-Frequency Branches

The effectiveness of symbolic execution originates from its ability to systematically enumerate the paths in a program, exposing vulnerabilities that may be hidden within it. The concolic executor typically solves constraints for the neighbor path of the seed execution path without determining the branch coverage state. This means that the concolic executor does not consider whether a particular branch has already been covered multiple times by previous test cases.

Most of the test cases generated by the fuzzer have the same path as the original seed, so most of them visit certain branches frequently. As a result, the concolic executor performs too many redundant constraint-solving attempts on the same branches, introducing significant unnecessary overhead and making it difficult to generate valuable test cases.

Figure 2 shows the coverage state of a branch in readelf that corresponds to the code fragment in Listing 1. This code snippet checks the number of program headers and outputs an error message if the number is illegal. During testing, the concolic executor read a test case from the fuzzer's queue that covered the left branch (indicated by the solid line in Figure 2), and then it tried to generate an input that would cover the right branch. However, by the time the concolic executor reached this branch position, the right branch had been executed by 197 test cases and the left branch had been executed by only 89 test cases. It can be seen that the right branch was executed about twice as often as the left branch,

indicating that the format of most of the test cases generated by the fuzzer was illegal. We can reduce the number of constraint-solving attempts for these high-frequency branches.

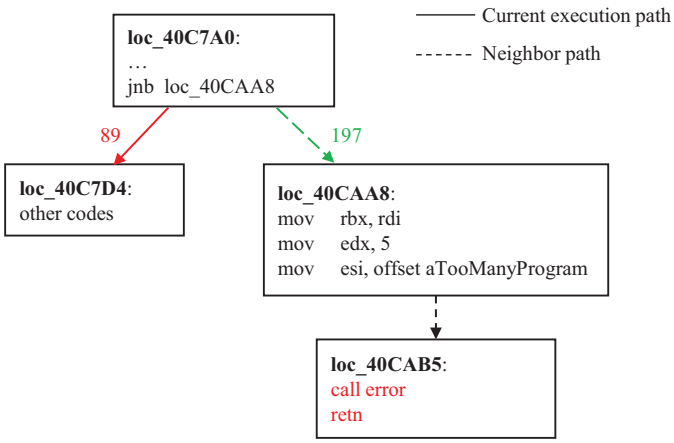

Figure 2. The branch coverage state of the branch in readelf in Listing 1.

Listing 1. An example of a conditional statement in readelf.

```
1 //@src/binutils/readelf.c:6031
2 if (filedata->file_header.e_phnum *
3     (is_32bit_elf ?
4       sizeof (Elf32_External_Phdr) :
5       sizeof (Elf64_External_Phdr))
6   >= filedata->file_size)
7 {
8   error(_("Too many program headers
9    -%#x-the file is not that big\n"),
10        filedata->file_header.e_phnum);
11  return false;
12 }
13 ... //other interesting code
```

3. Design

In this section, we describe the key components of BSFuzz. Figure 3 shows an overview of BSFuzz's architecture. Our improvements are based on QSYM [2] and AFL [8]. The method of concolic execution with dynamic instrumentation uses the design of QSYM [2].

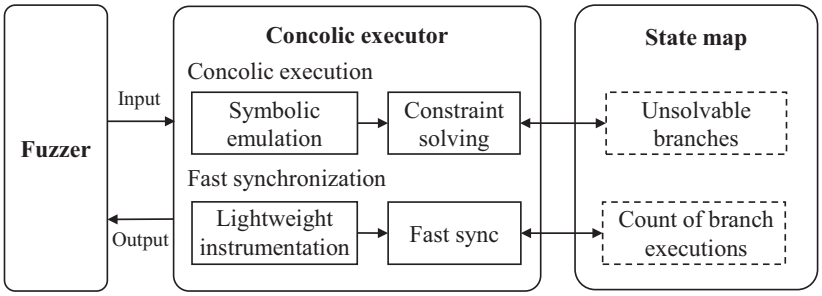

Figure 3. Overview of BSFuzz's architecture.

First, we add a *state map* with a size of 65,536 and 16 bits per entry to record the number of executions for each conditional jump. This map allows us to keep track of branch executions. BSFuzz improves performance by quickly synchronizing the latest

coverage state from the fuzzer to reduce constraint solving. Secondly, during concolic execution, we mark the branch locations that cannot be solved by concolic execution with a maximum value. BSFuzz then selectively executes constraints of interest. If the current branch is marked as *unsolvable*, it is skipped. This significantly reduces redundant constraint solving. BSFuzz also determines whether a branch is a high-frequency execution branch based on the number of executions recorded in the *state map*. If it is a low-frequency branch or appears in a new context, constraints solving is performed; otherwise, the branch is skipped.

3.1. Fast Synchronization

The latest fuzzing coverage state can serve as a good guide for concolic execution. By having access to this state, the concolic executor can focus its efforts on generating inputs for branches that have not yet been covered by the fuzzer's inputs. To record the location of unsolvable branches and the specific execution counts for the branches, we create a *state map* with an entry size of 16 bits.

We emulate the execution of the test cases using the Intel Pin tool. Pin instruments the instructions of the program at runtime, allowing us to track the execution traces of the program. During dynamic instrumentation, we only track the execution traces and do not consider the register value changes before and after the instructions. The advantage of this is that we can execute the test cases quickly and keep up with the speed of fuzzing. We only instrument the conditional jump instructions in binary to obtain the address of the instruction and the jump target address. Since the main concern of constraint solving is the constraints on these conditional jump locations, instrumenting only conditional jump instructions also minimizes the overhead.

3.2. Unsolvable Branch Filter

As described in Section 2.2, the concolic executor may perform repeated solving for some *unsolvable* locations. This can result in a significant waste of resources and reduce the overall efficiency of the concolic execution process. To address this issue, we developed a method for labeling *unsolvable* branch locations. If both the complete path and the last constraint in the path are *unsolvable*, we consider this location to be probabilistically related to the execution environment and therefore label this branch location. Listing 2 shows an unsolvable branch of the file program. The true branch condition of the *if* statement is that the `malloc` function returns a null pointer, indicating that memory allocation has been unsuccessful. This is usually not possible because the return value of malloc is NULL only when there is insufficient memory.

Listing 2. BSFuzz's branch coverage count.

```
1 h = hashPC(pc, taken)
2 state_map[(prev_jcc >> 1) ^ h]++
3 prev_jcc = h
```

We use two 16-bit large numbers to record the current state of each branch. If the branch is considered *unsolvable* with high probability, it is labeled with 65,534 in the *state map*. If the branch has already been covered by test cases, the last bit of the branch label is set to 1, resulting in a value of 65,535. When the program is executed again at a labeled branch location and tries to solve the current path constraint, it checks whether the branch has been labeled. If the branch is already labeled as *unsolvable*, it is not considered a branch of interest and will not be solved. This allows BSFuzz to avoid spending time and resources on branches that are unlikely to yield useful results. We call this strategy BSFuzz-uns.

We chose 65,534 and 65,535 as labels because fast synchronization shares the same bitmap with the *unsolvable* branch filter, but in fast synchronization, the number of test cases executing the same branch usually does not reach 65,534 or 65,535. Even if the number does reach or exceed 65,534, since neither the high-frequency branches nor the *unsolvable*

branches are of interest for constraint solving, they will be filtered out anyway and will not affect BSFuzz's judgment.

3.3. Frequency-Based Branch Prioritization

The scheduling mechanism is a crucial component of the parallel model of the fuzzer and the concolic executor. An effective scheduling mechanism can fully utilize the fuzzing and concolic execution and reduce redundant exploration. Our goal is to make better use of the coverage state from the fuzzer to guide the concolic execution process.

We generally let the concolic executor compute constraints on low-frequency branches and reduce the computation of constraints on high-frequency branches. The same branch at the same location will be treated as two different branches if the execution path is different from the previous one. The number of test case executions on all branches is recorded in the *state map*, and the count is increased once when a test case is executed. If the value of branch br_i in the *state map* is greater than the average *avg*, we consider it a high-frequency branch. The average *avg* is defined as

$$avg = \frac{\sum_{br_i \in S} N(br_i)}{|S|} \tag{1}$$

where S is the set of branches that have been covered and $N(br_i)$ is the number of test cases that cover branch br_i.

BSFuzz uses lightweight instrumentation in its synchronization process. In some situations, branches in different contexts may be treated as the same branch, leading to frequency summation. Therefore, BSFuzz maintains a call stack for the current execution, using hash values to distinguish between different contexts. If a branch is *solvable* and high-frequency but in a different context, its path constraint will be solved; otherwise, it will not. If a branch is low-frequency, its path constraint will be solved. This strategy is called BSFuzz-fre.

Algorithm 1 shows the whole process of branch filtering. First, *unsolvable* branches are excluded (line one). Then, branches are filtered according to their execution frequency (line four) and contextual environment (line six) to reduce the constraint solving for high-frequency branches.

Algorithm 1 Branch Filtering

Input: A branch *br*.
Output: Whether the branch is worth solving.

```
1:  if isUnsolvableBranch(br) then
2:      return false
3:  end if
4:  if isLowFrequencyBranch(br) then
5:      return true
6:  else if NewContext(br) then
7:      return true
8:  else
9:      return false
10: end if
```

4. Implementation and Evaluation

We implemented a prototype system called BSFuzz based on AFL [8] and QSYM [2]. AFL [8] is the most popular fuzzer and we used it to fuzz the programs. QSYM [2] is one of the best-performing hybrid fuzzers, and we used it for concolic execution, collecting constraints in the path and solving them. To prove the effectiveness of our method, we conducted extensive experiments with several target programs. Our experiment was designed to answer the following questions:

- **RQ1:** How effectively does fast synchronization keep up with the fuzzer's speed (Section 4.1)?
- **RQ2:** How effective is the filtering of unsolvable branches and high-frequency branches in improving the efficiency of hybrid fuzzing (Sections 4.2 and 4.3)?

Baseline. We compared our approach to QSYM [2]. When running experiments with QSYM [2], each program was assigned one CPU core for fuzzing and two CPU cores for concolic execution. Each independent BSFuzz experiment also used three cores: one core for fuzzing, one core for concolic execution (BSFuzz-ce), and one core for fast synchronization (Fast sync). Since the experiments for the unsolvable branch filter strategy did not require fast synchronization, we used only two cores for testing this strategy; i.e., one core for fuzzing and one core for concolic execution.

Benchmarks. To verify the validity of BSFuzz, we chose the latest versions of real-world programs for evaluation. These programs are frequently tested in fuzzing-related papers and use tcpdump, xpdf, libpng, and binutils. The configuration information and program versions are displayed in Table 2. Our initial seed files were from AFL [8] and UNIFUZZ [9].

Experimental setup. All experiments were run on an Ubuntu 16.04 LTS system equipped with four Intel(R) Xeon(R) Gold 5117 CPUs (each with 14 2.00 GHz cores) and 128 GB of RAM. We used up to three cores for each experiment.

Table 2. Configuration information and versions of the benchmark programs.

Program	Version	Input Format
readelf -a @@	binutils-2.40	elf
tcpdump -e -vv -nr @@	tcpdump-4.99	pcap
pdftotext @@ /dev/null	xpdf-4.04	pdf
pdfimages @@ /dev/null	xpdf-4.04	pdf
pdftops @@ /dev/null	xpdf-4.04	pdf
pngfix @@	libpng-1.6.40	png

4.1. Fast Synchronization

To demonstrate the speed of fast synchronization, we conducted a 24 h experiment with the programs in Table 2 using BSFuzz. During the experiment, we recorded the number of test cases processed by the concolic executor and the fast synchronization process, respectively. We compared the speed of both in terms of the number of test cases processed in 24 h. Figure 4 shows the results of the 24 h experiment. The left side represents the total number of test cases generated by the fuzzer (AFL), the middle side represents the number of test cases processed by BSFuzz's concolic executor (BSFuzz-ce), and the right side represents the number of test cases processed by the fast synchronization process (Fast sync). BSFuzz-ce and Fast sync processed test cases from the same fuzzer's queue. In addition, the processing speed of QSYM [2] for these programs is recorded in Table 1.

In the 24 h experiment, the fuzzer could generate thousands of test cases. On average, it took a few seconds to discover a test case with increased coverage. For example, programs using xpdf could generate nearly 20,000 test cases to increase coverage, while pngfix could generate only about 3000 test cases. The speed of BSFuzz-ce in executing most programs did not differ significantly. BSFuzz-ce could execute about 1000 test cases in 24 h; i.e., it could process a test case synchronized from the fuzzer's queue in an average of 80 s. It mainly performs symbolic emulation and constraint solving.

Fast synchronization is much faster than BSFuzz-ce in processing test cases and it was able to process all the test cases generated by the fuzzer in the 24 h experiment. Its disadvantage is that it has to execute test cases one by one, so it lagged behind the fuzzer at the beginning of the experiment. However, after fuzzing entered the deeper code area, the rate of generating meaningful test cases also gradually slowed down, at which point the progress of the fast synchronization was consistent with that of the fuzzer. As shown

in Figure 5, the fast synchronization mechanism was able to gradually catch up with AFL during the 24 h test. For pngfix, in particular, the fast synchronization mechanism was on par with AFL from the very beginning. In summary, the fast synchronization mechanism can synchronize the test cases in the fuzzer-generated queue in real time to obtain the latest conditional jump coverage state during concolic execution.

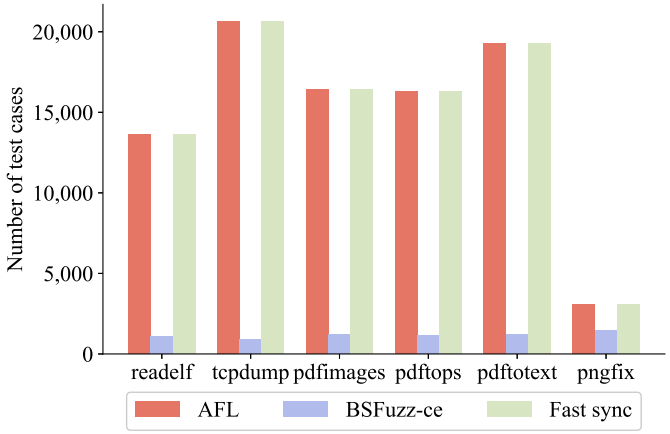

Figure 4. The numbers of test cases executed by the fuzzer (AFL), concolic executor, and fast synchronization in 24 h.

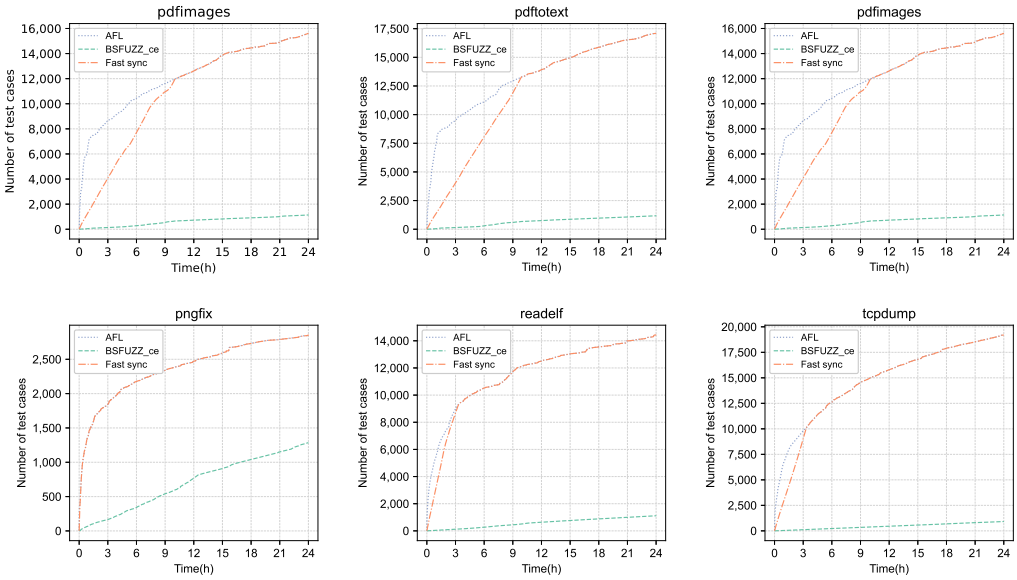

Figure 5. The numbers of test cases processed by different components over time with real-world programs.

4.2. Unsolvable Branch Filter

To demonstrate the performance improvement of the unsolvable branch filtering strategy, we used benchmark programs to compare the branch solving of QSYM and BSFuzz. Table 3 shows the branch-solving performance of each program over a 24 h period. The branch-solving states were classified as *solvable*, *partially solvable*, and *unsolvable*. "Processed branches" refers to all branches that the symbolic execution engine attempted to solve, regardless of whether or not they were successfully solved. "Solvable branches" are those branches for which the complete constraints could be solved successfully. "Unsolvable branches" are those branches where even the last constraint could not be satisfied. "Solving times" means the total number of times the symbolic execution engine performed branch solving. "Unsolved times" refers to the number of times the symbolic execution engine

failed to solve the last constraint. We primarily counted the total number of branches processed by the concolic executor, the number of solvable branches, and the number of unsolvable branches in 24 h. We also calculated the growth rate of BSFuzz relative to QSYM. BSFuzz significantly increased the total number of branches processed, with an average increase of nearly 50%. The number of branches successfully solved by BSFuzz also increased significantly, with an average growth rate of 22.87%.

Table 3. Comparison of the numbers of branches solved and total solving times for QSYM [2] and BSFuzz.

Program	Fuzzer	Processed Branches		Solvable Branches		Unsolvable Branches		Total Solving Times			Unsolved Times		
		Number	Growth	Number	Growth	Number	Growth	Number	Growth	Multiples	Number	Ratio	Multiples
pdftotext	QSYM	945	-	252	-	480	-	26,568	-	28.11×	11,614	43.71%	24.20×
	BSFuzz	1338	+41.59%	314	+24.60%	712	+48.33%	14,625	−44.95%	10.93×	712	4.87%	1×
pdftops	QSYM	786	-	230	-	348	-	25,293	-	32.17×	10,096	39.92%	29.01×
	BSFuzz	1283	+63.23%	316	+37.39%	683	+96.26%	13,849	−45.25%	10.79×	683	4.93%	1×
pdfimages	QSYM	809	-	243	-	380	-	26,185	-	32.37×	10,541	40.15%	27.67×
	BSFuzz	1119	+38.32%	247	+1.65%	584	+53.68%	11,041	−57.83%	9.87×	584	5.29%	1×
tcpdump	QSYM	4254	-	846	-	1016	-	24,592	-	5.78×	5709	23.21%	5.62×
	BSFuzz	6570	+54.44%	1165	+37.71%	1632	+60.63%	21,803	−11.34%	3.31×	1632	7.46%	1×
readelf	QSYM	2302	-	1205	-	299	-	36,259	-	15.75×	840	23.22%	28.16×
	BSFuzz	3115	+35.32%	1415	+17.43%	545	+82.27%	149,462	312.21%	47.98×	545	0.36%	1×
pngfix	QSYM	1438	-	219	-	547	-	38,586	-	26.83×	11,376	29.48%	20.80×
	BSFuzz	2027	+40.96%	223	+1.83%	595	+8.78%	28,590	−25.91%	14.10×	595	2.08%	1×
Average	QSYM	1756	-	499	-	512	-	29,581	-	16.85×	9622	32.53%	18.80×
	BSFuzz	2575	+46.68%	613	+22.87%	792	+54.76%	39,895	+34.87%	15.49×	792	1.98%	1×

In addition to comparing the number of branches processed, we also compared the change in the number of solving attempts. For most programs, the total solving attempts over 24 h decreased somewhat due to the fact that redundant solving of unsolvable branches was no longer performed. However, for readelf, the solving attempts increased by a factor of about three, suggesting that the unsolvable branch filtering allowed the concolic executor to spend more time solving the solvable branches of readelf. As discussed in Section 4.3, BSFuzz showed a more significant improvement with readelf. For all other programs, we effectively reduced the number of repeated solving attempts for the same branch location. For unsolvable locations, in particular, the ratio of unsolvable executions to the total number of constraint-solving attempts was greatly reduced. The average ratio decreased from 32.53% to 1.98%. The instances of unsolvable executions were equal to the number of unsolvable branches. This meant there were no more redundant unsolvable attempts. However, during QSYM's test process, the number of instances of unsolvable attempts was dozens of times higher than the number of unsolvable branches.

In summary, BSFuzz's unsolvable branch filtering strategy can effectively reduce the time wasted on unsolvable branches and use that time to solve solvable branches, effectively improving the efficiency of the concolic executor.

4.3. Code Coverage Effectiveness

To evaluate the effectiveness of BSFuzz, we conducted a series of experiments to compare its code coverage with that of QSYM [2] with the same benchmark program (Table 2). We also conducted separate experiments with BSFuzz-fre and BSFuzz-uns to demonstrate the contribution of each strategy to code coverage. For a fair comparison, we evaluated coverage using AFL's edge coverage (i.e., bitmap coverage). Each experiment was conducted for 24 h with each benchmark program and repeated five times. Since fuzzing is an uncertain process with unstable results, we repeated the experiments several times and used the average of the results for comparison. This is a common practice with fuzzers [9–11], with experimenters typically choosing 24 h as the runtime.

Figure 6 shows the average coverage of the programs achieved with different strategies over 24 h of repeated experiments. As can be seen in Figure 6, BSFuzz and its separate strategies BSFuzz-uns and BSFuzz-fre performed well in most cases and were able to cover

more code faster than QSYM [2]. In particular, BSFuzz clearly outperformed QSYM [2] with readelf. With pdftops and pngfix, BSFuzz-fre performed better than BSFuzz, mainly because BSFuzz-uns did not have a significant advantage with these programs. We believe this was because BSFuzz-uns filters out some critical branches that are important for achieving high coverage.

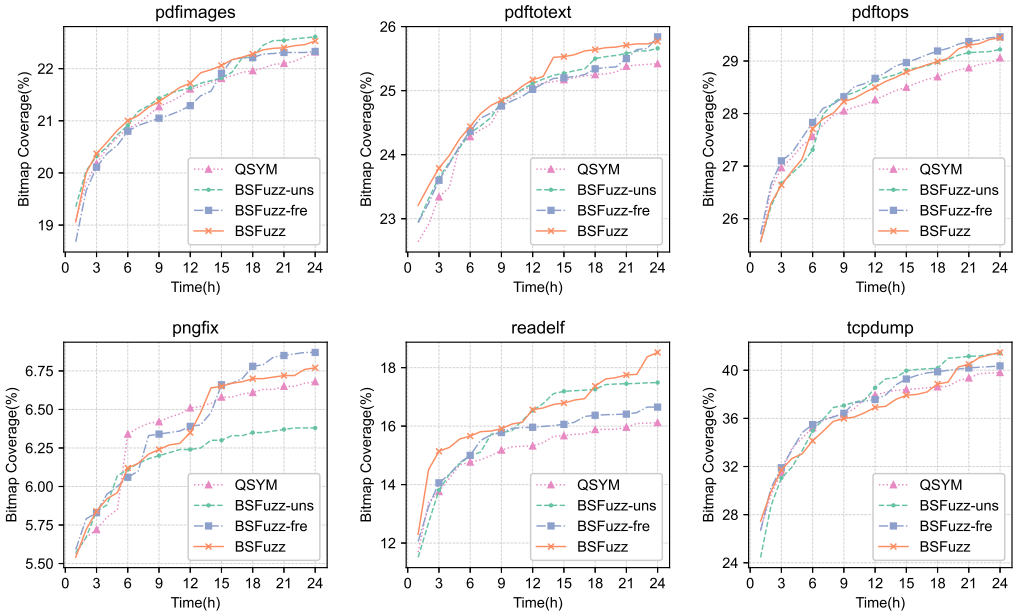

Figure 6. The bitmap coverage and time evaluation with real-world programs.

According to Table 4, BSFuzz demonstrated a higher coverage compared to QSYM [2]. Specifically, BSFuzz-uns, BSFuzz-fre, and BSFuzz had average growth rates of 2.32%, 1.35%, and 3.57%, respectively. In summary, BSFuzz's strategies enhance the speed of QSYM's [2] concolic executor. The effectiveness of these two strategies varies across different programs. Some programs may benefit more from one strategy than the other. Firstly, based on the data presented in Figure 6, we evaluated the efficacy of the BSFuzz-fre strategy. It was observed that, for the programs originating from xpdf, the fast-synchronization mechanism exhibited a slower processing speed and required approximately 10 h to process the test cases generated by AFL and catch up with AFL's progress. Therefore, BSFuzz-fre demonstrated inferior performance with pdfimages and pdftops compared to other programs. Analyzing the data in Table 3, we can see that the number of *solvable* branches for pngfix only increased slightly under the BSFuzz-uns strategy, which did not help pngfix explore more new *solvable* branches. Therefore, the BSFuzz-uns strategy was not effective with pngfix.

Table 4. Average bitmap coverage of BSFuzz and QSYM [2] in 24 h.

Program	QSYM	BSFuzz-Uns	BSFuzz-Fre	BSFuzz
pdftotext	25.42%	25.66% (+0.94%)	25.84% (+1.65%)	25.77% (+1.38%)
pdftops	29.18%	29.22% (+0.14%)	29.39% (+0.72%)	29.44% (+0.89%)
pdfimages	22.32%	22.61% (+1.30%)	22.33% (+0.04%)	22.53% (+0.94%)
tcpdump	39.81%	41.41% (+4.02%)	40.36% (+1.38%)	41.48% (+4.19%)
readelf	16.12%	17.49% (+8.50%)	16.65% (+3.29%)	18.52% (+14.89%)
pngfix	6.68%	6.38% (−4.49%)	6.84% (+2.40%)	6.77% (+1.35%)
Average	23.26%	23.80% (+2.32%)	23.57% (+1.35%)	24.09% (+3.57%)

5. Related Work

5.1. Coverage-Based Fuzzing

Intuitively, the more code a fuzzer is able to cover, the more likely it is to find vulnerabilities in the program being tested. Therefore, coverage-based fuzzers aim to cover more target code regions through various methods. For example, some fuzzers generate well-formed test cases to produce high coverage inputs [12]. Others provide more accurate coverage information to explore the program state more broadly [13–15]. These techniques can help to improve the chances of finding vulnerabilities in the program being tested.

Coverage tracking is generally achieved through program instrumentation. American Fuzzy Lop (AFL) [8] is representative of coverage-based fuzzers. It obtains edge coverage information by instrumenting the source code or binary using the QEMU mode and then mutates inputs to obtain new coverage. Other instrumentation techniques, such as Pin [2,16], Intel PT [17,18], DynamoRIO [19,20], E9Patch [21], and RetroWrite [22], are also widely used in practice. For example, BSFuzz uses Pin [2,16] to obtain coverage of conditional jump branches.

Fuzzers can use coverage information to select more useful seeds for fuzzing. By prioritizing seeds that are likely to increase coverage, fuzzers can improve their chances of finding vulnerabilities. Some fuzzers, such as Zeror [23] and UnTracer [24], only track test cases that increase coverage. Others, such as TortoiseFuzz [25], focus on test cases that cover edges associated with sensitive memory operations. Some fuzzers determine the priority of seeds based on the priority of execution paths [13,26–29]. BSFuzz prioritizes low-frequency branches based on their branch frequency for constraint solving, assigning a higher priority to branches with lower frequency and solving path constraints for low-frequency branches first.

5.2. Concolic Execution

Fuzzers can generate large numbers of test cases quickly but may struggle to generate inputs that satisfy complex conditions, such as judgments of magic bytes [30,31]. This is where concolic execution comes in. Concolic execution combines concrete and symbolic execution and uses constraint solving to pass certain branches in the program being tested. Common tools for concolic execution include Angr [32], KLEE [33], S2E [34], UC-KLEE [35], etc.

Given the effectiveness of concolic execution, many tools use it to solve various problems. BORG [36] guides concolic execution to point out potentially vulnerable spots in the program, thus triggering the buffer overread bug. SYMFUZZ [37] uses concolic execution to detect correlations between input bits and return a recommended mutation ratio that optimizes the fuzzing parameters.

Theoretically, concolic execution can cover all paths of a program. However, in practice, it has a well-known problem known as path explosion. While this problem cannot be resolved completely, there are many different methods that can be used to mitigate its impact. For example, Woodpecker [38] prunes redundant paths to speed up symbolic execution. Matryoshka [39] focuses only on conditional statements associated with the target branch. MergePoint [40] mitigates overhead by switching between dynamic and static symbolic execution. Eclipser [41] selects only a portion of the comparison instructions to generate approximate path constraints. BSFuzz utilizes fuzzing to explore most paths and directs the concolic executor to focus on solvable and low-frequency branches.

5.3. Hybrid Fuzzing

Hybrid fuzzing combines the strengths of fuzzing and concolic execution to improve the effectiveness of testing. The fuzzing is responsible for exploring most of the program's paths, while concolic execution solves the hard paths. Driller [1] invokes concolic execution to direct the execution path to a new region when the fuzzer gets stuck. Dowser [42] builds a guided fuzzer through taint tracking and program analysis, identifying code in the program that accesses arrays in loops (usually associated with buffer overflows) and symbolizing only the input bytes that affect the array index.

Despite the benefits of hybrid fuzzing, the slow speed of concolic execution can still cause inefficiencies. To alleviate this problem, QSYM [2] utilizes dynamic binary translation (DBT) to tightly integrate symbolic emulation and native execution. It achieves instruction-level symbolic emulation and significantly speeds up concolic execution. In addition, it has been found that there is an over-constrained problem with concolic execution, so QSYM generates more potentially useful inputs by partially solving the path constraints.

In addition, better cooperation between the fuzzer and the concolic executor is crucial in hybrid fuzzing. The creators of DigFuzz [3] identified the drawbacks of Driller's "demand launch" strategy. DigFuzz uses the Monte Carlo-based probabilistic path prioritization model (MCP^3) to calculate the probability of executing a specific path and then prioritizes the paths to ensure that concolic execution only works on the most difficult parts of the program. Since the fuzzer's mutation strategy is random, it may destroy the valid part of the input generated by the concolic executor. Based on this problem, PANGLOLIN [11] utilizes "polyhedral path abstraction" to limit the fuzzer's mutations to a given range so that the newly generated test cases still satisfy the corresponding path constraints. The creators of MEUZZ [7] believe that simple and fixed heuristics (e.g., selecting the seed with the smallest size) cannot be adapted to all situations, so they utilized machine learning to achieve adaptive seed scheduling, which can be extended to different programs. SHFuzz [43] also schedules the inputs based on the complexity of the branch, branch coverage information, and other factors.

Our method is orthogonal to the methods mentioned above and can be combined with other hybrid fuzzers. Unlike current hybrid fuzzers, we record whether constraint solving is successful or not, and we demonstrated that filtering out unsolvable positions improves concolic execution efficiency. Our method is similar to DigFuzz [3] in that both record the number of times each branch is executed. However, while DigFuzz uses the number of executions to calculate the difficulty of each path, we use it to reduce the constraint solving for high-frequency branches. In addition, both SHFuzz [43] and BSFuzz are concerned with the existence of a speed divide between the fuzzer and the concolic executor. SHFuzz [43] mitigates this problem through input scheduling, and BSFuzz mitigates it through the fast synchronization mechanism.

6. Discussion

Threats to validity. Although BSFuzz's fast synchronization mechanism is a lightweight dynamic instrumentation process, it requires a separate process to synchronize and record the coverage information for the test cases. Therefore, the fast synchronization mechanism increases CPU resource usage. During the experiments, the same number of cores was used for each experiment to fairly compare our approach with the others. To verify the effectiveness of our method, we used several benchmark programs that are commonly used in fuzzing papers. However, our method was not as effective with some programs. This was mainly because our method mainly improves the speed of concolic execution but does not improve the ability to solve constraints.

Limitations. We instrumented only conditional jump instructions in the program to improve the execution speed of the fast synchronization mechanism and to reduce the instrumentation overhead. In complex programs, there are often multiple paths that go through the same branch. If only the execution frequency for the branches is recorded, it may result in an inability to distinguish the frequency associated with different paths. Therefore, a finer granularity for path coverage would provide better accuracy for frequency recording. We chose branch coverage for the following reasons: Firstly, we prefer to quickly collect coverage information through lightweight instrumentation without focusing on the specific execution of branches (such as current register values). Therefore, we chose to only instrument at branches after weighing the pros and cons. Additionally, since concolic execution only performs constraint solving at branches, instrumenting branches allows our index to remain consistent with the index from concolic execution. Although we may confuse frequencies from different paths, we can still solve them from other branches of

these paths. Therefore, branch coverage does not have a significant negative impact on our final results.

7. Conclusions

In this paper, we proposed BSFuzz, a hybrid fuzzer designed to address the shortcomings of existing concolic executors. We found that these executors do not fully utilize the coverage states of all test cases or the history states of constraint solving. To address this issue, BSFuzz uses lightweight dynamic instrumentation to implement a fast synchronization mechanism that quickly executes test cases in the fuzzing queue. This allows the concolic executor to be guided by the latest coverage state obtained from the fuzzer and reduces constraint solving for high-frequency branches. In addition to this fast synchronization mechanism, BSFuzz also records locations that cannot be solved by the concolic executor and avoids attempting to solve these locations in the future. Our experimental results demonstrate that filtering both unsolvable branches and high-frequency branches can successfully improve the efficiency of hybrid fuzzing.

Author Contributions: Conceptualization, Q.H. and W.C.; Methodology, Q.H., W.C., Z.W. and S.L.; Validation, X.L. and X.K.; Formal analysis, Q.H. and W.C.; Investigation, Q.H. and W.C.; Data curation, Q.H., W.C. and Y.N.; Writing—original draft, Q.H. and W.C.; Writing—review & editing, Q.H., W.C. and Z.W.; Visualization, Q.H. and W.C.; Supervision, S.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Tianjin Key R&D Program (20YFZCGX00680), and the 2019 Tianjin New Generation AI Technology Key Project (19ZXZNGX00090).

Data Availability Statement: All data generated or analysed during this study are included in this published article.

Acknowledgments: The authors would like to thank anonymous reviewers for feedback and the support provided by the Tianjin Key R&D Program and the 2019 Tianjin New Generation AI Technology Key Project.

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

References

1. 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 2016 Network and Distributed System Security Symposium, San Diego, CA, USA, 21–24 February 2016. [CrossRef]
2. 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.
3. Zhao, L.; Duan, Y.; Yin, H.; Xuan, J. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In Proceedings of the 2019 Network and Distributed System Security Symposium, San Diego, CA, USA, 24–27 February 2019. [CrossRef]
4. Jiang, L.; Yuan, H.; Wu, M.; Zhang, L.; Zhang, Y. Evaluating and Improving Hybrid Fuzzing. In Proceedings of the IEEE/ACM International Conference on Software Engineering, Melbourne, Australia, 14–20 May 2023; pp. 410–422. [CrossRef]
5. Majumdar, R.; Sen, K. Hybrid Concolic Testing. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; pp. 416–426. [CrossRef]
6. Sen, K.; Marinov, D.; Agha, G. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Softw. Eng. Notes* **2005**, *30*, 263–272. [CrossRef]
7. Chen, Y.; Ahmadi, M.; Farkhani, R.M.; Wang, B.; Lu, L. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, 14–15 October 2020; Egele, M., Bilge, L., Eds.; USENIX Association: Berkeley, CA, USA, 2020; pp. 77–92.
8. Zalewski, M. American Fuzzy Lop. 2015. Available online: https://lcamtuf.coredump.cx/afl/ (accessed on 15 May 2023).
9. Li, Y.; Ji, S.; Chen, Y.; Liang, S.; Lee, W.H.; Chen, Y.; Lyu, C.; Wu, C.; Beyah, R.; Cheng, P.; et al. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, Vancouver, BC, Canada, 11–13 August 2021; pp. 2777–2794.
10. Peng, H.; Shoshitaishvili, Y.; Payer, M. T-Fuzz: Fuzzing by Program Transformation. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 697–710. [CrossRef]

11. 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. [CrossRef]
12. Kargén, U.; Shahmehri, N. Turning Programs against Each Other: High Coverage Fuzz-Testing Using Binary-Code Mutation and Dynamic Slicing. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 782–792. [CrossRef]
13. 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. [CrossRef]
14. 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. [CrossRef]
15. Nagy, S.; Nguyen-Tuong, A.; Hiser, J.D.; Davidson, J.W.; Hicks, M. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, 15–19 November 2021; pp. 351–365. [CrossRef]
16. Luk, C.K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; Wallace, S.; Reddi, V.J.; Hazelwood, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* **2005**, *40*, 190–200. [CrossRef]
17. Google. Honggfuzz. 2010. Available online: https://honggfuzz.dev/ (accessed on 10 June 2023).
18. Schumilo, S.; Aschermann, C.; Abbasi, A.; Wör-ner, S.; Holz, T. Nyx: Greybox Hypervisor Fuzzing Using Fast Snapshots and Affine Types. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, Vancouver, BC, Canada, 11–13 August 2021; pp. 2597–2614.
19. Google. DynamoRIO. 2014. Available online: https://github.com/DynamoRIO/dynamorio (accessed on 10 June 2023).
20. Fratric, I. WinAFL. 2016. Available online: https://github.com/googleprojectzero/winafl (accessed on 10 June 2023).
21. Duck, G.J.; Gao, X.; Roychoudhury, A. Binary Rewriting without Control Flow Recovery. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, London, UK, 15–19 June 2020; pp. 151–163. [CrossRef]
22. Dinesh, S.; Burow, N.; Xu, D.; Payer, M. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1497–1511. [CrossRef]
23. Zhou, C.; Wang, M.; Liang, J.; Liu, Z.; Jiang, Y. Zeror: Speed Up Fuzzing with Coverage-Sensitive Tracing and Scheduling. In Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 21–25 September 2020; pp. 858–870.
24. Nagy, S.; Hicks, M. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 787–802. [CrossRef]
25. Wang, Y.; Jia, X.; Liu, Y.; Zeng, K.; Bao, T.; Wu, D.; Su, P. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In Proceedings of the 2020 Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2020. [CrossRef]
26. Lemieux, C.; Sen, K. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 475–485. [CrossRef]
27. Yan, S.; Wu, C.; Li, H.; Shao, W.; Jia, C. PathAFL: Path-Coverage Assisted Fuzzing. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, 5–9 October 2020.
28. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Trans. Softw. Eng.* **2019**, *45*, 489–506. [CrossRef]
29. Yue, T.; Wang, P.; Tang, Y.; Wang, E.; Yu, B.; Lu, K.; Zhou, X. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), 2020, USENIX Association, Berkeley, CA, USA, 12–14 August 2020; pp. 2307–2324.
30. Li, Y.; Chen, B.; Chandramohan, M.; Lin, S.W.; Liu, Y.; Tiu, A. Steelix: Program-State Based Binary Fuzzing. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), Paderborn, Germany, 4–8 August 2017; pp. 627–637. [CrossRef]
31. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In Proceedings of the 2017 Network and Distributed System Security Symposium, San Diego, CA, USA, 26 February–1 March 2017. [CrossRef]
32. 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; pp. 138–157. [CrossRef]
33. Cadar, C.; Dunbar, D.; Engler, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08), San Diego, CA, USA, 8–10 December 2008; pp. 209–224.
34. Chipounov, V.; Kuznetsov, V.; Candea, G. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI), Newport Beach, CA, USA, 5–11 March 2011; pp. 265–278. [CrossRef]

35. Ramos, D.A.; Engler, D. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15), Washington, DC, USA, 12–14 August 2015; pp. 49–64.
36. Neugschwandtner, M.; Milani Comparetti, P.; Haller, I.; Bos, H. The BORG: Nanoprobing Binaries for Buffer Overreads. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 2–4 March 2015; pp. 87–97. [CrossRef]
37. Cha, S.K.; Woo, M.; Brumley, D. Program-Adaptive Mutational Fuzzing. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 725–741. [CrossRef]
38. Cui, H.; Hu, G.; Wu, J.; Yang, J. Verifying Systems Rules Using Rule-Directed Symbolic Execution. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems—ASPLOS'13, Houston, TX, USA, 16–20 March 2013; pp. 329–342. [CrossRef]
39. Chen, P.; Liu, J.; Chen, H. Matryoshka: Fuzzing Deeply Nested Branches. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19), London, UK, 11–15 November 2019; pp. 499–513. [CrossRef]
40. Avgerinos, T.; Rebert, A.; Cha, S.K.; Brumley, D. Enhancing Symbolic Execution with Veritesting. *Commun. ACM* **2016**, *59*, 93–100. [CrossRef]
41. Choi, J.; Jang, J.; Han, C.; Cha, S.K. Grey-Box Concolic Testing on Binary Code. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 736–747. [CrossRef]
42. Haller, I.; Slowinska, A.; Neugschwandtner, M.; Bos, H. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In Proceedings of the 22nd USENIX Conference on Security (SEC'13), Washinton, DC, USA, 14–16 August 2013; pp. 49–64.
43. Mi, X.; Wang, B.; Tang, Y.; Wang, P.; Yu, B. SHFuzz: Selective Hybrid Fuzzing with Branch Scheduling Based on Binary Instrumentation. *Appl. Sci.* **2020**, *10*, 5449. [CrossRef]