Hindawi
Security and Communication Networks
Volume 2021, Article ID 1987844, 15 pages
https://doi.org/10.1155/2021/1987844

Research Article

StFuzzer: Contribution-Aware Coverage-Guided Fuzzing for Smart Devices

Jiageng Yang, Xinguo Zhang, Hui Lu, Muhammad Shafiq, and Zhihong Tian

Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, China

Correspondence should be addressed to Zhihong Tian; tianzhihong@gzhu.edu.cn

Received 20 July 2021; Revised 21 August 2021; Accepted 30 August 2021; Published 20 September 2021

Academic Editor: Muhammad Ahmad

The root cause of the insecurity for smart devices is the potential vulnerabilities in smart devices. There are many approaches to find the potential bugs in smart devices. Fuzzing is the most effective vulnerability finding technique, especially the coverage-guided fuzzing. The coverage-guided fuzzing identifies the high-quality seeds according to the corresponding code coverage triggered by these seeds. Existing coverage-guided fuzzers consider that the higher the code coverage of seeds, the greater the probability of triggering potential bugs. However, in real-world applications running on smart devices or the operation system of the smart device, the logic of these programs is very complex. Basic blocks of these programs play a different role in the process of application exploration. This observation is ignored by existing seed selection strategies, which reduces the efficiency of bug discovery on smart devices. In this paper, we propose a contribution-aware coverage-guided fuzzing, which estimates the contributions of basic blocks for the process of smart device exploration. According to the control flow of the target on any smart device and the runtime information during the fuzzing process, we propose the static contribution of a basic block and the dynamic contribution built on the execution frequency of each block. The contribution-aware optimization approach does not require any prior knowledge of the target device, which ensures our optimization adapting gray-box fuzzing and white-box fuzzing. We designed and implemented a contribution-aware coverage-guided fuzzer for smart devices, called StFuzzer. We evaluated StFuzzer on four real-world applications that are often applied on smart devices to demonstrate the efficiency of our contribution-aware optimization. The result of our trials shows that the contribution-aware approach significantly improves the capability of bug discovery and obtains better execution speed than state-of-the-art fuzzers.

1. Introduction

Various smart devices have been installed in many situations. The security of smart devices is essential to ensure the normal functions of these smart devices. The root cause of any attack is the potential vulnerabilities in smart devices. Finding potential bugs in smart devices is the most important factor to ensure the security of smart devices. There are many automatic approaches to identify bugs in applications or devices, such as taint, symbolic execution, or fuzzing. Fuzzing is the most effective vulnerability identification technology, which inputs various random data to the target application to anticipate a dangerous execution state. If the test case crashes the target application, a bug must be triggered. However, fuzzing mainly recognizes bugs in the shallow logic of an application, because modern applications usually require complex input formats that are not conducive to exploring deeper execution states. Fuzzing randomly mutates an input file that would destroy the crucial data structure of the input file. The mutated file as a test case will cause the target application to stop in a shallow execution state.

State-of-the-art coverage-guided fuzzing such as AFL [1] utilizes edge coverage to select test cases which triggered more interesting execution states. If a test case triggered a new branch, it will be regarded as an interesting seed. The natural branch coverage mechanism thinks any branch of a target application has the same contribution. However, branches that belong to deep logic are much more difficult to trigger than branches that exist in shallow logic [2]. Especially, in the smart device, the execution of these devices involves multiple sensors and arguments, that the execution

must be satisfied by special sensors or argument. In other words, triggering branches in deep logic contributes more than branch being triggered in shallow logic. However, AFL-family cannot consider the contribution of different branches. In detail, fuzzing that employs branch coverage preserves all test cases which triggered new branches as seeds. These seeds will be used to generate new test cases in next iteration. Unfortunately, AFL cannot distinguish branches which have more interesting states. Therefore, AFL has no effective strategy for seed selection. In order to optimize seed selection strategy of coverage-guided fuzzing, there are two problems should be solved. First, we should distinguish the contributions of different branches according to the depth of a branch. Second, we should select test cases with greater contributions for the next fuzzing iteration.

There are many works related with seed selection strategy. Rebert et al. [3] proposed that good seed selection strategies can improve the speed of fuzzing and increasing the chance of recognizing vulnerabilities, even reducing the runtime and memory overhead of fuzzing. AFLFast [4] found low-frequency paths through Markov chain and prioritized seed triggered these paths. AFLFast exposed several previously unreported CVEs that could not be exposed by AFL in 24 hours. The seed selection strategies of AFLFast increase the exploration frequencies of low-frequency paths successfully. VUzzer believed that deeper code blocks are more difficult to be triggered and prioritized seeds exercising longer path. VUzzer found more crashes than AFLPIN [5], which indicated that exploring deeper paths may increase the probability of finding crash. Good seed selection strategies can improve the ability to find bugs, but it is often difficult to determine the precise contributions of various test cases.

In this paper, we present StFuzzer, a contribution-aware coverage-guided fuzzing which obtains precise contributions of all branches and guides fuzzing to explore deeper execution states. Comparing with other seed selection solutions such as VUzzer, our work provides a solution based on the contribution information that combines static and dynamic information, which considers the accurate control flow information and dynamic runtime information. This combination scheme can help tremendously to improve the efficiency of path exploration and avoid unnecessary drilling. The crucial intuition of our work is that deeper basic blocks can contribute more to exploring interesting execution states. The most important question is how to calculate the depth of each basic block. We defined the depth of basic blocks based on the control flow feature of target applications and modify the depth information according to the runtime feature of fuzzing process. The control flow feature of the target application will be captured by the customized static analysis. Our solution dynamically collects useful runtime information to evaluate the value of different paths. These static analysis and dynamic information guide the fuzzing process to deeper code region and to trigger more interesting execution states. Due to this novel combination solution in seed selection, StFuzzer could find more crashes than AFL on various real-world applications. Meanwhile, our solution reduces the useless exploration of shallow logic, and StFuzzer reproduced the same crashes faster than AFL.

Contribution. This paper makes the following contributions:

(1) We propose a novel contribution-aware seed selection solution that combines the accurate control flow information of static analysis and dynamic runtime information to evaluate the contributions of each branch.

(2) We design a sophisticated contribution calculation algorithm that statically assesses the value of various branches. And this algorithm considers the influence of the loop structure, which is unresolved in similar solutions.

(3) We implement our approach to build a fuzzer tool named StFuzzer based on AFL and evaluate StFuzzer on 4 real-world applications running on smart devices.

(4) StFuzzer found 5 zero-day vulnerabilities and already reported them to corresponding vendors.

2. Background and Related Work

In this section, we present the background of coverage-guided fuzzing. Firstly, we introduce the main workflow of coverage-guided fuzzing. Secondly, we describe the details and limitations of seed selection and seed mutation in fuzzing loop.

2.1. Coverage-Guided Fuzzing. Fuzzing is an automatic software testing technique that attempts to input random data into the target application and expects the target has exceptions. If the fuzzing process captures an exception, it means that a vulnerability had been triggered by a test case. The fuzzing technique can be classified as generation-based and mutation-based. The generation-based fuzzing relies on grammar constraints to generate random inputs, such as Peach [6]. Since generation-based fuzzers must require grammar constraints of corresponding input format, the usage of generated fuzzing is not universal. Mutation-based fuzzers mutate seeds of the initial corpus to generate test cases which are put into the target application for execution. Mutation-based fuzzers do not require configuration files for different target application, which makes this technology more versatile.

According to different exploration strategies, mutation-based fuzzing can be classified into directed fuzzing and the coverage-guided fuzzing. Directed fuzzing will explore more specific code regions based on the strategy used. Directed gray-box fuzzing [7] requires researchers provide deterministic strategies for specified types of vulnerabilities. Therefore, directed fuzzing is available for finding defined bugs, but it is unfair for unknown bugs. Coverage-guided fuzzing is the most popular technology for finding vulnerabilities. Coverage-guided fuzzing determines the contribution of test cases based on code coverage. The intuition of coverage-guided fuzzing is that the greater the coverage, the greater the chance of triggering bugs. Figure 1 shows the

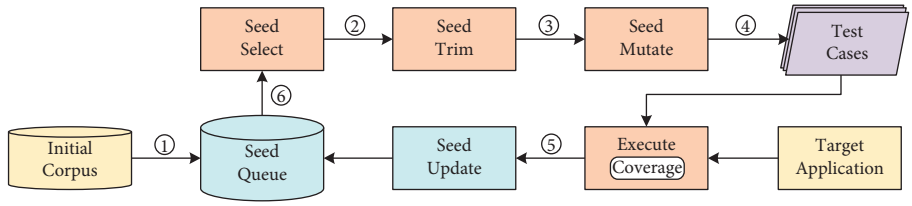

Figure 1: The main workflow of coverage-guided fuzzing.

main workflow of coverage-guided fuzzing. It usually contains six stages as follows: (1) it reads the initial corpus to load original test cases into the seed queue, (2) scheduling higher-priority seeds from seed queue as input files, (3) trimming input files to the smallest size and coverage of the trimmed file is the same as the file before trimming, (4) mutating the input file through various heuristic mutation algorithms to generate the test case, (5) monitoring coverage triggered by test cases and persisting test cases with new coverage, and (6) going to Step 2 and starting the next loop iteration.

Steps 2 to 6 build a complete fuzzing loop iteration. The core components include seed selection, seed mutation, and execution monitor. Any improvements of these components would promote the ability of coverage-guided fuzzing to find vulnerabilities. In following sections, we will analyze and discuss the core components of the fuzzing loop in detail.

2.2. Seed Selection. Seed selection module schedules more interesting seeds according to some runtime features determined by the employed policies. Some studies demonstrated that good seed selection algorithms can greatly increase the number of bugs found. Rebert et al. proposed that good seed selection strategies can improve the speed of fuzzing and reduce the runtime and memory overhead of fuzzing process. AFLFast found low-frequency paths through Markov chain and prioritized seed triggered these paths. VUzzer believed that deeper basic blocks are more difficult to be triggered and prioritized seeds which traversed deeper blocks. Meanwhile, VUzzer observed that the most of generated test cases will end with error handling code and deprioritized seeds trapped into error handling code. CollAFL [8] proposed two selection policies which drive the fuzzer towards nonexplored paths. CollAFL prioritized seeds that invoke more memory access operations or paths have more nonexplored neighbor branches. Wang et al. [9] proposed a seed scheduling algorithm to support the multilevel coverage metric based on the multiarmed bandit model, which archives higher code coverage.

Other seed selection solutions attempt to guide the fuzzing to explore specific vulnerable code regions [10, 11]. QTEP [12] presented a quality-aware seed prioritization technique, which leveraged the statistic defect prediction model to defect fault. ProFuzzer [13] proposed an on-the-fly probing technique that discovered the relations between input bytes and program behaviors, which utilizes the information to select more powerful seeds. GREYONE [14] utilized lightweight dynamic taint analysis to evaluate the constraint conformance on all tainted untouched branches, and prioritizing seeds which could reach these branches. FIFUZZ [15] proposed a context-sensitive SFI-based approach to guide fuzzing exploring error handling code.

There are ample works on seed selection, and the result of these studies indicates that seed selection is crucial for coverage-guided fuzzing. However, when we reproduced some trials through these open source works, we found two problems. On the one hand, the implementation of some studies is inadequate. The implementation proposed by the work cannot exactly perform the corresponding selection policy. For example, as aforementioned, VUzzer proposed prior exploring deeper block policy. The solution of VUzzer measured the depth of basic blocks through the Markov probabilistic model, but it simply assigns a fixed probability of 1 to each backedge belonging to the loop structure. In general, the loop structure has a great influence on the depth of a basic block. In a word, its seed selection algorithm does not consider the loop structure in the application, resulting in inaccurate results of the fuzzing process. On the other hand, implementing some policies requires not only static compile-time information but also dynamic runtime information. For example, VUzzer assigned an equal probability to all basic blocks' outgoing edges. Remarkably, it is no possible that outgoing edges of a basic block have the same probability to be executed. The probability of a branch should be calculated by statistics. Therefore, the imprecise probability makes VUzzer's result incorrect. In conclusion, the accuracy of path features has a negative impact due to the imperfect implementation.

Following the observations, we propose a novel contribution mechanism, which incorporated the influence of the control flow and the dynamic statistics at the runtime. Our solution considers the accurate control flow information which solves the challenge caused by nested branches in various loop structures. And we also convert the triggered frequencies of basic blocks during the iterations of the fuzzing process into the contribution of each block. Compared with other solutions, the contribution mechanism measures the distances of all basic blocks for the program exploration, which is more accurate to seeds with high quality. And the contribution mechanism does not require prior knowledge of the target application that is appropriate for all applications.

2.3. Seed Mutation. Seed mutation module mutates seeds by multiple mutating algorithms and generates new test cases for the next loop iteration. Different mutation algorithms insert or replace special characters in the seed for different situations and attempt to traverse all the traps within inputs.

For example, AFL mutates seeds through four random mutation algorithms: bitflip, arithmetic, interest, and havoc.

However, there are some challenges in the target application, which restrain the fuzzing process to explore more code region. For example, most applications have a variable number of sanity checks. These sanity checks involve intricate inspections or require unique input formats, which are difficult to satisfy through arbitrary mutations. Good seed mutation strategies need to answer two questions: where to mutate and how to mutate.

Many solutions to both of these questions have been proposed. Program transformation-based approaches disable sanity checks to help the fuzzing process to discover more execution logic. MutaGen [16] proposed that mutating the code of generating programs and leveraged available information about the input format encode in the generated programs to produce high-coverage test cases. T-Fuzz [17] attempted to strip relevant code that hinders program explosion. However, changing the normal logic will introduce more unintended consequences, leading to more false positive bugs. And transformation-based approaches have to suffer trimming explosion in complex applications, which the blind mutation engine is limited for program explosion.

Hybrid fuzzing utilizes symbolic execution to solve complex branch which cannot be satisfied through simple mutation engine. However, symbolic execution or concolic execution produces huge amount of runtime overhead and memory overhead, which have hindered the efficiency of fuzzing. Ma et al. [18] studied an approach to automatically find program executions which reach a particular line. Driller [19] used concolic execution to produce test cases when the fuzzer getting stuck, and got higher coverage. However, driller has a terrible execution speed and works with a low efficiency. The best way to mitigate the performance overhead is proposing a lightweight symbolic execution engine. QSYM [20] designed a fast concolic execution engine and implemented instruction-level symbolic emulation to overcome performance bottlenecks of the concolic executor. Some works believed that reducing the calls of symbolic execution will significantly mitigate overhead. Digfuzz [21] proposed a probabilistic path prioritization model to estimate the probability of exercising each path and prioritized them for concolic execution. Peng et al. [22] leveraged a distance-based directed fuzzing and a dominator-based directed symbolic execution mechanism to discover 1-day vulnerabilities in binary patches. SAVIOR [23] prioritized the concolic execution of the seeds which can uncover more vulnerabilities. Lee et al. [24] proposed a constraint-guided directed fuzzing based on the target site and the data conditions. However, it is difficult to evaluate whether a seed uncovers more vulnerabilities if lacking priori knowledge.

Some fuzzers depended on the result of static or dynamic analysis, such as honggfuzz [25] and VUzzer. Honggfuzz recognized operands of CMP-like instructions at runtime to insert into the seed. VUzzer leveraged control- and data-flow features of the application to consider interesting factors for mutation. Steelix [26] proposed a program-state based approach, which utilized lightweight static analysis to provide comparison progress information to infer magic bytes efficiently. Learn and Fuzz [27] attempted to capture the structure of well-formed inputs and utilize well-structured seed-based deep learning. Rajpal et al. designed DNN (deep neural network) solution to predicate which bytes in the seed to mutate. FuzzGuard [28] utilized the learned model to predict a given input if shortened the seed distance to improve the effectiveness of directed fuzzing. Some works [28–34] designed sematic models to optimize mutation or identify potential vulnerabilities or attacks based on special target platforms. Liang et al. [35] proposed a secure decision tree classification for online diagnosis services on different platforms. However, according to evaluation of trials, the result of various lightweight static analysis has a slight effect on program explosion. Meanwhile, the effects of learning-based optimizations are also unclear, because of the uninterpretable results of learning.

As aforementioned, multiple optimizations for seed mutation involve symbolic or concolic execution, static analysis, dynamic taint, or machine learning. The hybrid fuzzing will be faced on the path explosion program and heavyweight performance overhead, which seriously reduce the effectiveness of the fuzzing process. The taint-assisted fuzzing also has numerous performance overhead and the overtaint or undertaint problems. The learning-based optimization is unclear or uninterpretable for fuzzing. In a word, existing optimizations cannot improve the effectiveness of seed mutation.

3. Calculation of Contribution

In a large-scale application, the control flow graph of the application may be very complicated, resulting in a low coverage of the fuzzing process. In the fuzzing process, most test cases traverse the simple logic of applications and the complex logic will be rarely triggered. Unfortunately, the more complex the logic, the more bugs. Because it is difficult to discover complex logic by the fuzzing process, simple logic is easier to be triggered, and a lot of time is spent exploring simple logic. The contribution of each basic block for exploring the whole application is different. Basic blocks in complex logic have more contribution than basic blocks in simple logic. The existing approach did not apply the contribution of basic blocks or thought that all basic blocks have the same contribution. In this paper, for each basic block, our solution gives quantified contributions based on control flow and execution flow. And we apply the contribution as an important factor for the fuzzing process.

3.1. Static Contribution of Basic Block. Due to different control flow structures, basic blocks of the target application have different execution probabilities. The entry block of a called function must be executed, so the execution probabilities of entry block are 1. We cannot accurately calculate the jumping probabilities of a branch that a basic block jumps to any successor blocks. Therefore, for a given basic block, we assign equal jumping probabilities to all its successor blocks. In other words, if a basic block B has n

successors, the jumping probabilities of any successors are $1/n$. As same as the transition of VUzzer, the execution probability of each basic block will be dependent on a probabilistic model called Markov process. The execution probability of a basic block hinges on the execution probability of its predecessor block and its own jumping probability. The execution probability of a basic block B is calculated as follows:

$$\text{prob}(B) = \sum_{pB \varepsilon \text{pred}(B)} \text{prob}(pB) * \text{prob}j(B), \quad (1)$$

where $\text{pred}(B)$ is the set of all the predecessors of B and the $\text{prob}j(B)$ is the jumping probability of B. In this way, we can regard the whole execution as a Markov chain to evaluate the contribution of each node in the CFG.

However, this solution is only applicable to the special case where the CFG is a directed acyclic graph and cannot deal with the explicit or implicit loop structures. To be specific, loop statements or jump statements would lead to cyclic structures in the CFG. The cyclic structure makes certain control flow within loop structures be repeatedly executed, which disrupts the calculation by the Markov chain. As aforementioned, the jumping probability of any block must be less than one. According to formula (1), the execution probability of the header block in a loop structure will become smaller and smaller with multiple iterations. The header block dominates all blocks of the loop structure, so the execution probability of the header block must become larger and larger with multiple iterations, which is contrary to the result of calculation through the Markov chain. Obviously, the execution probability inferred by the Markov chain is unsuitable to the real consequence.

Considering the above problems, we utilize mathematical expectation to estimate the static contribution feature of loop structures instead of execution probabilities. In loop structures, the execution probability of a basic block is an integral multiple of the execution probability within a separate iteration. Since the header block dominates other basic blocks in the loop, the execution probability of other blocks is similar to the execution probability of the corresponding header block. The execution probability of normal blocks can be calculated according to the mathematical expectation of the header block. And the mathematical expectation of the header block conforms that the time of continuous states staying obeys geometric distributing. When it is assumed that entering a loop means the failure of a geometric distributing and exiting a loop means the success of a geometric distributing, the distribution of loop structures is consistent with the geometric distribution. If we assign a certain probability to a Bernoulli experiment, the mathematical expectation will be calculated in the geometric distribution. Therefore, calculating the mathematical expectation of a loop structure relies on the success probability of loop iterations. Since the success probability of a Bernoulli experiment for a loop is the same as the execution probability of the corresponding header block, the mathematical expectation of a loop is consistent with the expectation of the header block. In summary, we can calculate the mathematical expectation of the header block through the success or failure probability of a loop iteration, and then we can infer expectations of all blocks in loops according to the Markov process.

Figure 2 shows the three typical loop structures in the control flow graph of the target application. Figure 2(a) shows that the control flow graph contains a single top-level loop structure. The basic block A is the entry block and the basic block E is the exit block. The loop consists of block B, block C, and block D. We can infer that the execution probability of block B and block D are both 1. Since the jumping probability of the backedge is 0.5, the success probability of corresponding geometric distributing is also 0.5. According to formula (2), the mathematical expectation of the loop is 2. As aforementioned, the mathematical expectation of block C or D is calculated through the Markov process, which indicates that the mathematical expectation is reasonable as static contribution in this case:

$$E(X) = \frac{1}{P(X)}. \quad (2)$$

In addition to the simple single loop structure, there are many nested loop structures in real-world applications. As Figure 2(b) shows, a loop is a subloop of a top-level loop, and we regard the nested loop structure as a nested geometric distribution. The feature of nested loop structures is that the top-level loop will be executed first and to jump into the subloop before exiting the top-level loop. After the subloop completes loop iterations, the execution flow will return to the top-level loop. In other words, for top-level loops, the failure probability of geometric distribution must be 1. Therefore, for the nested loop structure, the crucial factor to calculate mathematical expectation is the failure probability of the subloop. We can easily calculate the expectation of the subloop's header block based on the failure probability of the subloop and then infer probabilities of all blocks according to the Markov process, until the execution flow exits the current subloop. After exiting the subloop, the mathematical distribution of top-level can be computed by the expectation of the corresponding header block. For example, as Figure 2(b) shows, the execution probability of basic block C is 0.5 and the jumping probability of the edge for C to F is 0.5, and we infer the execution probability of basic block F is 0.25. As a result, the failure probability of the subloop is 0.5:

$$E(X) = \frac{1}{P(X)} * P(Y). \quad (3)$$

According to formula (3), the mathematical expectation of the subloop is 1, and the mathematical expectation of corresponding header block C is assigned a value of 1. The jumping probability of the edge from C to D is 0.5, the mathematical expectation of D is 0.5 based on the Markov process. Thus, the probability of returning the top-level loop from the subloop (i.e., the jumping probability of the edge from D to B) is 0.5. Consequently, the success probability of the top-level loop is 0.5. According to formula (2), the mathematical expectation of the top-level loop is 2, which means the mathematical expectation of basic block C is 2. In

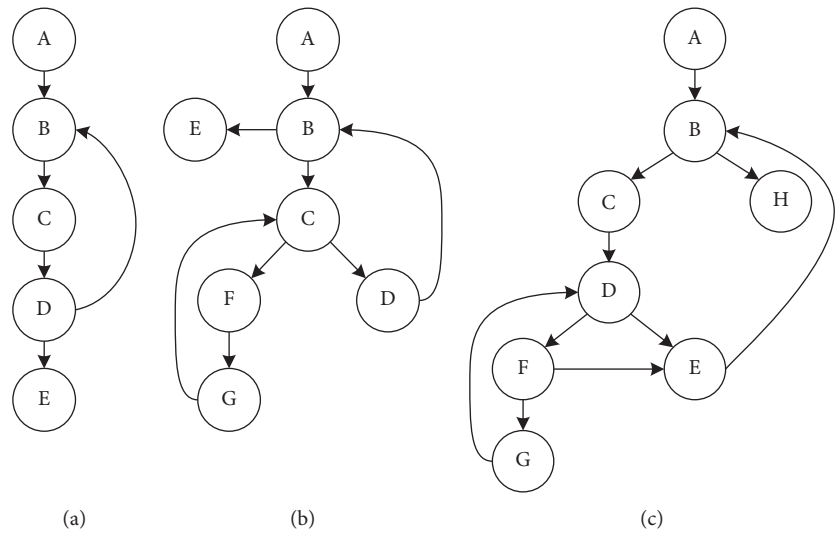

Figure 2: Three typical loop structures. (a) The CFG includes a single top-level loop. (b) The CFG includes a nest loop structure. (c) The CFG includes a loop containing a jumping statement.

total, we have calculated that the mathematical expectation of C's successors (block D and block F) is both 1.

Figure 2(c) shows that the loop structure contains jumping statements (for example, break and continue) that may abort the loop iteration and jump out of the loop. In this case, the control flow may not complete an entire loop iteration and exit the loop before the header block, which apparently increases the failure probability of the corresponding Bernoulli trial for the loop. The failure or success probability of this loop structure could be inferred through the execution probability of the header block and then correctly calculate the mathematical expectation of other basic blocks. As Figure 2(c) shows, in the subloop that is composed of basic blocks D, F, and G, a branch from block F to block E jumps out of the subloop, leading to the end of the iteration. The execution probability of block C and D are both 0.5, and the jumping probability of the edge from block D to block F is 0.5; thus, the execution probability of block F is 0.25. According the Markov process, the execution probability of block G is 0.125. As same as the subloop in Figure 2(b), we calculate the success probability of the subloop is 0.75.

As formula (3) shows, the mathematical expectation of the corresponding header block D is 0.67, as the expectation of the subloop is also 0.67. The mathematical expectation of the exit block F is 0.33, and the jumping probability of the edge from F to G is 0.5; thus, the expectation of block G is 0.17, and the expectation of block E is 0.5. The success probability of the top-level loop is 0.5, and then the mathematical expectation of the top-level is 2 based on formula (2). To sum up, we have calculated that the expectation of block D is 1.33 and the expectation of block F is 0.67.

According to the above three loop structures, we can calculate the mathematical expectation of any loop structure as follows:

$$E(\mathrm{Loop}_i) = \frac{E(\mathrm{header}_i)}{E(\mathrm{header}_i) - E(\mathrm{header}_i) * \mathrm{Prob}_{\mathrm{jump}}(\mathrm{backedge}_i)}, \tag{4}$$

where $E(\mathrm{header_i})$ is the mathematic expectation of the corresponding header block in Loop_i, and the $\mathrm{Prob}_{\mathrm{jump}}(\mathrm{backedge}_i)$ stands for the jumping probability of the back edge from any exiting block to the header block header_i. Based on the expectation of the loop, we infer the expectation of any block in the loop through the Markov process.

As aforementioned, we propose a static contribution calculation approach based on the probability of geometric distribution and mathematical expectation, which leads the complex loop structure that brings about nested control flow jumps. Combined with Markov process, this approach can accurately calculate the contribution of all basic blocks in any control flow graph.

Algorithm 1 demonstrates the process of calculating the mathematical expectation of each block in any loop structure. It first traverses all successors of any visited basic block (line 1), and calculate the mathematical expectations of visited blocks based on the execution probability and the jumping probability of the edge between visited blocks (line 3~5). Then, we correctly process the jumping probability of the backedge as the above three cases and eliminate the visited backedge to convert the loop into a directed acyclic graph (line 7~9). Last, we traverse the rest of visited blocks and add these blocks into the analysis queue (line 11~15).

3.2. Dynamic Contribution of Basic Block. We assume that each basic block has the same execution probability when calculating the static contribution. Successors of the same basic block have the same jumping probability. However, in the real-world application, jumping probabilities of different

```
Input: B: the set of visited basic blocks
B: the set of visited basic blocks
Output: Calculating the expectation of each basic blocks
(1) if ∀bεB, ∃cε Succ (b) ∩ cεB then
(2)   if jumpweight (b, c) ≥ 0 then
(3)     temp = weight (c) ÷ (Weight (c) − jumpweight (b, c) × weight (c)
(4)     for each block ρε pred (succ (b)) do
(5)       jumpweight (p, c) = temp × jumpweight (p, c)
(6)     end for
(7)     jumpweight (b, c) = −1
(8)     remove (B, Succ (c))
(9)     add (U, c)
(10)  else
(11)    add (B, b)
(12)  end if
(13) else
(14)  add (U, c)
(15)  add (B, b)
(16) end if
```

Algorithm 1: Calculate expection of basic blocks in a loop structure.

successors for the same branch are different. For example, in a loop, the jumping count of the branch from the latch block to the header block and the jumping count of the branch from the latch block to the exiting block satisfy the following equation:

$$\frac{C_H}{C_E} = \text{Iter}, \tag{5}$$

where C_H is the jumping count of the branch from the latch block to the header block, and C_E is the jumping count of the branch from the latch block to the exiting block; Iter is the number of iterations. If we only assume each successor of any branch has the same jumping probability, the calculated static contribution is inaccurate. Therefore, our approach contains not only static contributions determined by control flow but also dynamic contributions inferred from runtime information. We count the number of executions to determine the triggered frequency of a discovered block, which will be used to evaluate the dynamic contribution.

However, there are two problems with simply counting the number of executions as a dynamic contribution of any block. First, in real-world applications, the sequential execution flow will lead to the nonuniform triggering probabilities of basic blocks. Different basic blocks achieve different functions, and there are fixed executed sequences of blocks for any function. Some basic blocks perform a shallow logic of the application, which may be executed frequently. Some other basic blocks complete a deep logic of the application and will only be triggered if all conditions in the path are satisfied. In our intuition, a basic block in deeper logic has a greater dynamic contribution because it is more difficult to trigger. Second, the number of iterations for the fuzzing process is numerous. There will be astonishing runtime overhead if we count the number of triggers for each block. Meanwhile, the number of discovered blocks will continue to be added as the fuzzing process is iterated. We have to traverse all discovered blocks in each iteration to calculate statistics, which causes more and more serious performance overhead. In summary, we consider the negative impacts caused by the above two problems when calculating the dynamic contribution. For the first problem, basic blocks in shallow logic will be executed more frequently than basic blocks in deep logic, and we utilize the discovery order of basic blocks to solve it. For the second problem, the more and more serious performance overhead as iterating, we set a threshold T of iterations to mitigate the performance overhead. We only record executions of discovered blocks within T iterations to evaluate the dynamic contribution of the discovered block.

If the number of fuzzing iterations is less than the threshold T, we accurately calculate the dynamic runtime probability based on the statistical data of each basic block as follows:

$$P_{\text{dyn}}(\text{block}_i) = \begin{cases} \dfrac{P_{\text{dyn}}(\text{block}_i) * x}{x+1}, & (\text{block}_i \text{ non\_executed}), \\ \dfrac{P_{\text{dyn}}(\text{block}_i) * x + 1}{x+1}, & (\text{block}_i \text{ executed}), \end{cases} \tag{6}$$

where $P_{\text{dyn}}(\text{block}_i)$ is the dynamic runtime probability of basic block i and the argument x stands for the number of the iterations.

If the number of fuzzing iterations is bigger than the threshold T, we simplify the calculation of the dynamic runtime probability as follows:

$$P_{\text{dyn}}(\text{block}_i) = \begin{cases} \dfrac{P_{\text{dyn}}(\text{block}_i) * (T-1)}{T}, & (\text{block}_i \text{ non\_executed}), \\ \dfrac{P_{\text{dyn}}(\text{block}_i) * (T-1)}{T}, & (\text{block}_i \text{ executed}). \end{cases} \tag{7}$$

Algorithm 2 shows how to calculate the dynamic contributions of basic blocks as the above approach. As aforementioned, there are two situations which are divided by the number of the fuzzing iterations (line 2 and line 8). In first case, we calculate the dynamic contribution of each block according to formula (6) (line 3~7). In second case, if the number of the fuzzing iterations is greater than the threshold T, we attempt to infer the dynamic contribution based on formula (7) (line 9~13).

4. Design and Implementation

In the previous section, we proposed an approach to infer the static and dynamic contributions of basic blocks through the control flow of the target application and the runtime information of fuzzing iteration. However, the contribution information cannot guide the fuzzing process to explore more interesting code regions, because the fuzzing process does not utilize contribution information in fuzzing iterations. Therefore, we design a contribution-aware fuzzer named StFuzzer and introduce the main work flow and the crucial implementation details of StFuzzer.

4.1. Design of StFuzzer. As aforementioned, we provide a contribution metric based on the control flow and runtime information, which quantifies the contribution of basic blocks for exploring applications. The approach only utilizes the control flow information of target applications and does not use sophisticated data flow analysis. So, this approach can be used for gray-box fuzzing and white-box fuzzing. In white-box fuzzing, the source code can be accessed, and we can accurately obtain the control flow information at compile-time to calculate the static contribution. In gray-box fuzzing, the source code cannot be accessed, and we decompile the target binary to obtain the control flow graph.

Figure 3 shows the main work flow of StFuzzer:

(1) If the target is a gray-box program, StFuzzer decompiles the target application.

(2) StFuzzer instruments the target application based on different instrumentation approaches according to whether the target application is gray-box or white-box.

(3) StFuzzer calculates the static and dynamic contribution of each basic block.

(4) After the calculation of static and dynamic contributions, the fuzzing process prioritizes favorable seeds based on seed selection strategy, which are mutated to generate fresh test cases.

(5) Executing new test cases in the target application, and updating test cases that trigger new coverage, and returning the execution information for the calculation of dynamic contributions.

(6) Go to 3.

Since static contributions of the target application are fixed, the static contribution calculation is only performed once when the fuzzing process accesses a target application for the first time. After the static contribution is calculated, the static contribution information of basic blocks will be stored in a map file in the form of <key, value>, where key means the ID of the basic block and the static contribution information as values to calculate the relative information. In the next iteration of StFuzzer, the fuzzing process will read the static contribution information from the map file into in-process memory. So, the calculation of static contributions does not increase the runtime performance overhead. StFuzzer performs dry run of all test cases to collect and calculate the dynamic contribution information and initializes all test cases in the seed queue. Then, we execute the test case in the target application and count the number of executions for each basic block. The number of executions is the basic information for calculating dynamic contributions of basic blocks. It is worth noting that the fuzzing process would affect dynamic contributions of different blocks, and the dynamic contribution information also has a crucial impact on the program exploration of the fuzzing process.

The state-of-art coverage-guided fuzzer AFL believes that the importance of all paths is the same. However, in the real-world application, some execution paths would trigger the core functions of the application, and some paths only call error handling function to end the execution. Obviously, the former plays a more important role in the fuzzing process. In our work, the importance of paths depends on the contributions of corresponding basic blocks. In order to implement the feedback loop between dynamic contributions and heuristic exploration, the contribution information would be applied to the evolutionary seed selection algorithm. StFuzzer calculates the weight of each test case according to the triggered blocks and corresponding static and dynamic contributions of these blocks and proposes a weighted seed selection algorithm.

The weighted seed selection algorithm is the most important module of StFuzzer, because it achieves the feedback in the fuzzing loop. The weighted seed selection algorithm selects seeds which trigger more basic blocks with higher contributions. We believe mutating these seeds has a greater probability of triggering more blocks with high contributions, which is conducive to improve the efficiency of application exploration and vulnerability discovery. It is worth noting that some basic block may be triggered multiple times by a test case in a fuzzing iteration. So, a block may have an excessive dynamic contribution. To avoid a basic block with the excessive contribution causing the fuzzing process to trap a special region of the target, we consider translating the static and dynamic contribution to the weight of a basic block as follows:

$$W\left(\text{block}_i\right) = \frac{\log\left(1/C_{\text{sta}}\left(\text{block}_i\right)\right)}{\log 2} + \frac{\log\left(1/C_{\text{dyn}}\left(\text{block}_i\right)\right)}{\log 2}, \tag{8}$$

where $C_{\text{sta}}(\text{block}_i)$ is the static contribution of block i, and $C_{\text{dyn}}(\text{block}_i)$ is the dynamic contribution of block i. The higher the contribution of a basic block, the higher the

Input: B the set of basic blocks
i: the number of the fuzzing iterations
T: Settings threshold
Output: Calculating the dynamic Contribution of each block
(1) for each block $bb\varepsilon B$ do
(2) if $i \leq \Gamma$ then
(3) if b has been executed then
(4) $C_{dyn}(b) = (C_{dyn}(b) \times (i-1)+1) \div i$
(5) else
(6) $C_{dyn}(b) = (C_{dyn}(b) \times (i-1)) \div i$
(7) end if
(8) else
(9) if b has been executed then
(10) $C_{dyn} = (C_{dyn}(b) \times (T-1)) \div T + \overline{T}$
(11) else
(12) $C_{dyn} = (C_{dyn}(b) \times (T-1)) \div T$
(13) end if
(14) end if
(15) end for

Algorithm 2: Calculate dynamic contributions of basic blocks.

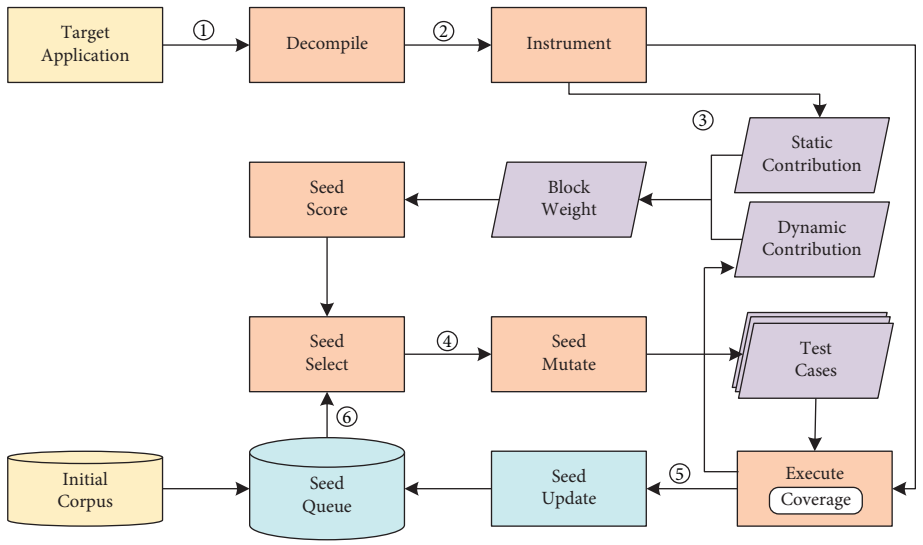

Figure 3: The framework of StFuzzer.

weight of the basic block. The score of a test case t is calculated by the weight of triggered blocks as follows:

$$S_t = \left(\sum_{\text{block}_i \varepsilon BB(t)} W(\text{block}_i) \right) \times |\text{BB}(t)|, \quad (9)$$

where $\text{BB}(t)$ stands for the set of triggered basic blocks by test case t and $|\text{BB}(t)|$ represents the number of the set $\text{BB}(t)$. The score of a test case considers not only the weight of blocks, but also the number of triggered blocks, which ensures the basic of coverage-guided fuzzing. In a word, the weighted seed selection algorithm of StFuzzer contains the block coverage and contributions of basic blocks, which will guide the fuzzing process to select seeds that trigger more blocks with higher contributions.

4.2. Implementation. We implemented a prototype of StFuzzer with about 5000 lines of C/C++ code and 400 lines of python code. The prototype of StFuzzer supports performing the target applications only with binary file or LLVM bitcode. We introduce some key implementation details of the prototype in this section.

4.2.1. Decompile and Analyze Target Binaries. The contribution-aware approach is applicable to analyze application with binary or source code. If there is only a binary file of the target application, StFuzzer needs to decompile the binary to acquire the control flow information. Therefore, we implemented an interface to decompile the binary file based on IDA-python to support both the gray-box target and white-box target.

4.2.2. Contribution Calculation. The core component of StFuzzer is the contribution calculation, which includes the static contribution and the dynamic contribution. For gary-box applications, we implemented a common interface to analyze the static contribution information according to the control flow graph which is constructed by the IDA decompiler and our analysis script. For white-box applications, we achieved a llvm module pass to calculate the static contribution information.

4.2.3. Seed Selection. Evolutionary fuzzers usually select favorable seed as the initial seed used to further mutations. The seed selection usually relies on a certain seed selection strategy. We implemented a module based on the seed scoring approach introduced in the above section to compute scores of all test cases and to prioritize test cases with higher score. The weighted seed selection algorithm can be used to the block coverage metric and the edge coverage metric. We implemented StFuzzer by the block coverage metric to adapt the black-box application. We also implemented a set of extensible interfaces to help StFuzzer utilizing the edge coverage metric in future improvement.

4.2.4. Mutation Engine. Mutation is a critical component for a coverage-guided fuzzer. A good mutation engine mutates the initial seed multiple times to traverse code region as much as possible. We implemented an extensible interface to allow StFuzzer to use other mutation engines quickly and support other mutation optimizations such as taint or symbolic execution.

4.2.5. Crucial Data Manager. StFuzzer has acquired more and more analysis results during the iteration of fuzzing, which provides a feedback to guide the fuzzing process to explore more basic block with higher contributions. The analysis results mainly include the fixed static contribution of each block and growing dynamic contribution of each discovered block. We implemented an extensible mechanism to store values of static contributions in a bitmap (the ID of each basic block as key), similar to the bitmap storing edge information of AFL. Since the dynamic contribution information is changed as iterations of fuzzing, we implemented an in-process model to store and index dynamic contributions in a list-based structure which is traversed fast.

5. Evaluation

In this section, we design a series of experiments to evaluate the efficiency and the effectiveness of bug finding of StFuzzer. And we compare StFuzzer with other fuzzers to show the code coverage and crash growth improvements in real-world applications.

5.1. Experiment Setup. Baseline fuzzers: we carefully choose several well-known coverage-guided fuzzers as baseline fuzzers, including AFL, QSYM, and honggfuzz. First, AFL is the most popular and most famous coverage-guided fuzzing. There are many AFL-family fuzzers that have been implemented based on AFL. Second, QSYM is the art-of-state symbolic execution assisted fuzzer, which designed a special symbolic execution engine to satisfy the fuzzing process. So, QSYM usually has higher coverage than other coverage-guided fuzzers. Meanwhile, the execution speed and efficiency of QSYM is better than other symbolic execution assisted fuzzer such as Driller. Third, honggfuzz is the core fuzzing engine of Google's OSS-fuzz project. Honggfuzz relies on the feature of control flow and the feature of data flow to achieve higher coverage than AFL. Therefore, we chose these fuzzers as baseline fuzzers to ensure the improvement of StFuzzer.

Target applications: we deliberated three factors when choosing the target applications, including functionality diversity, code scale, and popularity. The target applications should be from real-world applications running on smart devices to make sure that StFuzzer is able to recognize zero-day vulnerabilities in real-world applications, not just perform on the dataset. Finally, we chose four popular open source applications, including ffjpeg, libredwg, catdoc, and libsolv. They have diverse functionalities, including popular image parser library (ffjpeg), industry designing library (libredwg), office file parser (catdoc), and operation system component (libsolv). These applications are supported by GNU and open source communities and get frequent maintenance. There are different scales of application, including large scales (libredwg, libsolv) and small scales (ffjpeg and libsolv).

Experiment environment: we run these baseline fuzzers with the same configuration that a virtual machine with two Intel CPU @ 2.40 GHz and 8 GB RAM, running on Ubuntu 18.04. In order to keep the same running of all fuzzers, we use the same shell arguments to run each fuzzer during 24 hours. The code coverage and execution speed of each experiment are compared to evaluate the capability of bug finding and performance overhead.

5.2. The Impact of Threshold T. Our dynamic contribution solution proposed the calculation with different stages to mitigate the performance overhead of calculating dynamic contributions. Specifically, StFuzzer calculates the dynamic contribution by formula (6) if the number of the fuzzing iteration is less than threshold T. We used formula (7) to calculate the dynamic contribution if the number of the fuzzing iteration is more than threshold T. The value of threshold T must influence the overhead of StFuzzer and the accuracy of the dynamic contribution. In this section, we assign different values to threshold T and discuss the variation of StFuzzer with different threshold T.

Table 1 demonstrates the value of block coverage triggered by StFuzzer in ffjpeg and catdoc. The first column represents the different value of threshold T. We have listed the max value in five experiments and the average value of coverage in catdoc and ffjpeg. In catdoc and ffjpeg, StFuzzer obtains the highest coverage when the threshold T is assigned a value of 100,000. For both of catdoc and ffjpeg, StFuzzer gets the lowest coverage when the threshold T is

Table 1: The block coverage triggered by StFuzzer.

Threshold T	catdoc		ffjpeg	
	Max	Average	Max	Average
100	316	312	211	205
1,000	290	288	207	203
10,000	306	300	215	204
100,000	339	322	226	211
1,000,000	291	280	213	209

Table 2: The number of execution paths triggered by StFuzzer.

Threshold T	catdoc		ffjpeg	
	Max	Average	Max	Average
100	199	190	231	223
1,000	172	168	253	238
10,000	199	186	276	259
100,000	223	208	279	277
1,000,000	166	152	238	231

1,000. In a word, the exploration capability of StFuzzer is the best if the threshold T is assigned a value of 100,000.

Table 2 shows the number of execution paths triggered by StFuzzer with different values of threshold T in ffjpeg and catdoc. As same as Table 1, the first column represents the different value of threshold T. In catdoc and ffjpeg, StFuzzer discovered the largest number of execution paths when the threshold T is assigned a value of 100,000. In catdoc, StFuzzer found the smallest number of execution paths when the threshold T is 1,000,000. For different situations in ffjpeg, StFuzzer found the smallest number of execution paths when the threshold T is 100. Comparing with the case of block coverage, the exploration ability of StFuzzer is the best when the threshold T is 100,000.

Table 3 shows the average speed of StFuzzer with different values of threshold T in ffjpeg and catdoc. As same as Table 1, the first column represents the different value of threshold T. The average speed stands for the number of executions in one second. In catdoc, the average speed reaches maximum when the threshold T is 100,000. By contrast, in ffjpeg, the average speed is the slowest when the threshold T is 100,000, but the difference in average speed is too small to be ignored.

Table 3: The average execution speed of StFuzzer.

Threshold T	catdoc		ffjpeg	
	Max	Average	Max	Average
100	334	322	76	72
1,000	635	622	58	55
10,000	632	631	44	42
100,000	670	662	44	41
1,000,000	423	418	46	43

5.3. Conclusion. The capability of bug finding and application exploring will be the best when the threshold T is set to 100,000. The influence of the threshold T on execution speed is limited. Therefore, StFuzzer has a balance of bug finding and runtime overhead if the threshold is set to 100,000. As a result, in the following experiment, we set the threshold T to 100,000 to attain the best performance of StFuzzer.

5.4. Code Coverage Evaluation. Code coverage is one of the most important factors to evaluate a coverage-guided fuzzer. In coverage-guided fuzzing, more code coverage means the more code region has been triggered, the greater the probability of triggering hidden bugs. Some work has demonstrated that the probability of bug discovery increases by about 0.92% when the code coverage increased by one percentage. The code coverage of a coverage-guided fuzzer directly indicates the vulnerability recognizing ability of this fuzzer. Thus, we evaluated baseline fuzzers and StFuzzer on 4 real-world applications for 24 hours to record the code coverage of each fuzzer.

The coverage improvement: it is worth noting that AFL-family fuzzers apply the edge coverage metric that causes many hash collisions in the bitmap that saved edge information. Since our approach calculates the contribution for each basic block, StFuzzer intuitively applies the block coverage metric. With the intention of comparing coverage of all fuzzers uniformly, we intended to transform the edge coverage to the block coverage (i.e., the number of discovered basic blocks).

Table 4 shows the average coverage of different fuzzers on the four applications within 24 hours. Honggfuzz performs the best in baseline fuzzers because it applies complex control flow features and data flow features. Comparing with AFL, StFuzzer triggered much more basic blocks. Especially, in the large-scale application, StFuzzer discovered 646.6% more blocks of libredwg and 250.7% more blocks of libsolv. StFuzzer outperforms the second best fuzzer, honggfuzz, in all applications, at least over 4.6%. In large-scale applications (libredwg and libsolv), StFuzzer discovered more blocks than honggfuzz at least over 6.5%.

Figure 4 shows the growth of block coverage over time. The trend of growth indicates that block coverage of AFL and QSYM has been rising for a while, but in a short period of time, it has reached the upper limit of block coverage much lower than honggfuzz and StFuzzer in these target applications. In all target applications, StFuzzer has attained a substantial increase in coverage at the beginning stage of the fuzzing and is usually faster than AFL and QSYM (based on the derivative of the coverage growth curve). In the beginning stage, the coverage growth curve of StFuzzer is similar to that of honggfuzz. However, after this stage, StFuzzer is able to discover more blocks than honggfuzz in the exploration stage. It demonstrates that our contribution-aware optimization has a better effect on program exploration than the optimization applied in honggfuzz.

5.5. Conclusion. StFuzzer could discover more blocks than all baseline fuzzers, which demonstrates the vulnerability recognizing ability of StFuzzer is better than that of other fuzzers. The improvement of vulnerability recognizing

Table 4: The average coverage triggered by various fuzzers.

Application	AFL	QSYM	honggfuzz	StFuzzer
catdoc	121 (+158.7%)	123 (+154.5%)	283 (+10.6%)	313
ffjpeg	91 (+125.3%)	79 (+159.5%)	196 (+4.6%)	205
libredwg	2334 (+646.6%)	3439 (+406.7%)	16366 (+6.5%)	17425
libsolv	2227 (+250.7%)	2260 (+245.5%)	7245 (+7.8%)	7809

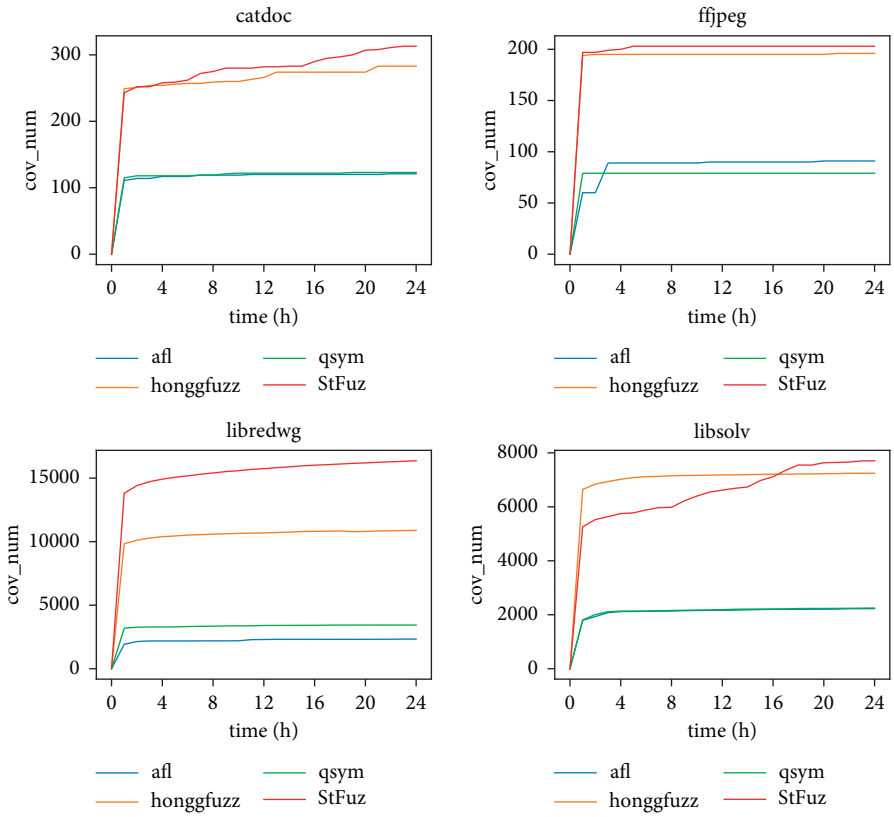

Figure 4: The growth of block coverage for each fuzzer.

ability is more obvious in small-scale applications such as catdoc and ffjpeg.

5.6. Execution Path Evaluation. Block coverage metric records the triggered blocks, but the execution information of block coverage is not enough accurate. Path coverage metric will track the order of all blocks, providing the most complete code coverage information. To be clearer, we count the number of seeds in the seed queue as path coverage, which is better to show the improvement of our contribution-aware optimization. In the fuzzing iteration, executing a test case will trigger an execution path. The more execution paths, the more triggered execution states of the target application. In most case, the vulnerability is only reproduced in a specific execution path or specific function call sequence. If a test case has triggered a new execution path, the test case will be stored in the seed queue. The more test cases in the seed queue stand for that the more execution states have been explored by the fuzzing process. Therefore, we used the number of test cases in the seed queue to evaluate the execution path triggered by different fuzzers.

Table 5 shows the number of triggered execution paths by different fuzzers. The first column stands for the different target applications. Comparing with AFL and QSYM, StFuzzer finds much more execution paths in these applications. Especially, in libredwg, StFuzzer found 1030.2% more execution paths than AFL. On average, StFuzzer found 15.9% more paths than the second best fuzzer, honggfuzz, found 327.9% more paths than AFL, and found 120.2% more paths than QSYM.

5.7. Conclusion. It demonstrated that StFuzzer outperforms all baseline fuzzers in terms of path exploration. StFuzzer can trigger more execution states of a target application within a certain period of time. The improvement of path exploration shows that the contribution-aware approach is beneficial to common coverage-guided fuzzing.

5.8. Zero-Day Vulnerability Evaluation. The basic functionality of any fuzzer is finding potential bugs or vulnerabilities in the target application. Therefore, it is the most

Table 5: The number of execution paths triggered by various fuzzers.

Application	AFL	QSYM	honggfuzz	StFuzzer
catdoc	80 (+158.8%)	55 (+276.4%)	143 (+44.8%)	207
ffjpeg	145 (+89%)	135 (+103%)	250 (+9.6%)	274
libredwg	1978 (+1030.2%)	13100 (+70.7%)	20932 (+6.8%)	22356
libsolv	8370 (+33.8%)	8578 (+30.5%)	10954 (+2.2%)	11195

direct indicator to reflect the efficiency of a fuzzer that whether the fuzzer finds zero-day vulnerabilities in target applications. We chose the number of zero-day vulnerabilities discovered as the crucial indicator of the last trial to evaluate the effectiveness of contribution-aware optimization for coverage-guided fuzzing.

Usually, a coverage-guided fuzzer will trigger the same bug multiple times during the loop iterating of fuzzing, because the fuzzing process repeatedly mutates a seed that triggered a bug. Duplicated crash samples disturb the exploitable analyzing process of the bug, which is undesirable for any security researchers. Since all target applications of our experiments are open source, we recompiled all target application with AddressSanitizer to filter out crash samples with the same root cause. In total, StFuzzer found 5 zero-day vulnerabilities in these applications. And we have submitted these vulnerabilities to corresponding vendors. These vulnerabilities have been confirmed by vendors and have been fixed in the latest version.

Table 6 shows the number of zero-day vulnerabilities discovered by different fuzzers. StFuzzer found 5 zero-day vulnerabilities, and the number of discovered vulnerabilities is more than that of all baseline fuzzers. It is worth noting that StFuzzer found a vulnerability in libredwg, and other fuzzers could not find this vulnerability. StFuzzer discovered 150% more vulnerabilities than the second best fuzzer, honggfuzz, which indicates that our contribution-aware optimization has a good effect on vulnerabilities recognition.

5.9. Conclusion. The contribution-aware optimization proposed in this paper helps the fuzzing process to identify more potential vulnerability. Compared with baseline fuzzers, StFuzzer identified much more zero-day vulnerabilities in these target applications.

5.10. Performance Overhead Evaluation. As aforementioned, the calculation of the static contribution for each basic block is performed before the first iteration of the fuzzing process. Calculating the static contribution does not introduce runtime performance overhead of the fuzzing process. However, the calculation of the dynamic contribution for each basic block relies on the runtime execution information of basic blocks discovered, which involves a dynamic calculating procedure. The calculation of the dynamic contribution causes some runtime overhead of the fuzzing process. We evaluated the performance overhead introduced by the additional calculation of the dynamic contribution according to the execution speed of each fuzzer.

Figure 5 shows the average execution speed of each fuzzer in these four applications. Since AFL is the simplest coverage-guided fuzzing and does not apply any high overhead optimization, we selected the average execution speed of AFL as the baseline in this trial. In this trial, we made an interesting discovery that the contribution optimization produces different performance overheads depending on the scale of the target application. In small-scale applications (catdoc and ffjpeg), StFuzzer has a higher execution effectiveness than AFL, i.e., the average execution speed of StFuzzer is slightly beyond that of AFL. StFuzzer achieved faster execution of each test case in consequence of our contribution-aware optimization guiding the fuzzing process to trigger more basic blocks with higher contributions within a fixed period. In small-scale applications, the performance benefit of the contribution-aware optimization exceeds the runtime overhead of the optimization. In above evaluations, honggfuzz has the best vulnerability recognition ability in the baseline fuzzers, which benefits from the honggfuzz's optimization that applied control flow features and data flow features. However, due to the enormous performance overhead brought by complex data flow analysis, honggfuzz acquires the worst execution speed. The data flow analysis of honggfuzz generates a lot of performance overhead even in small-scale applications. Comparing with honggfuzz, the contribution-aware optimization of StFuzzer is more useful.

Table 6: The number of zero-day vulnerabilities found by various fuzzers.

Application	AFL	QSYM	honggfuzz	StFuzzer
catdoc	1	0	1	2
ffjpeg	0	0	0	0
libredwg	0	0	0	1
libsolv	0	1	1	2

In large-scale applications (libredwg and libsolv), the average execution speed of StFuzzer is slower than that of AFL. It means that the performance overhead for calculating dynamic contributions has increased significantly. In large-scale applications, the core logic of the applications is complex, which leads AFL trapped into the shallow logic and the exploration ended before drilling the core functions. Therefore, AFL and QSYM have a high execution speed in large-scale applications. Comparing with honggfuzz, the average execution speed of StFuzzer is increased by 40% and 116% in libredwg and libsolv, respectively.

5.11. Conclusion. The contribution-aware optimization of StFuzzer is a lightweight optimization solution, which introduces less performance overhead. In small-scale applications, since the performance benefit of our optimization,

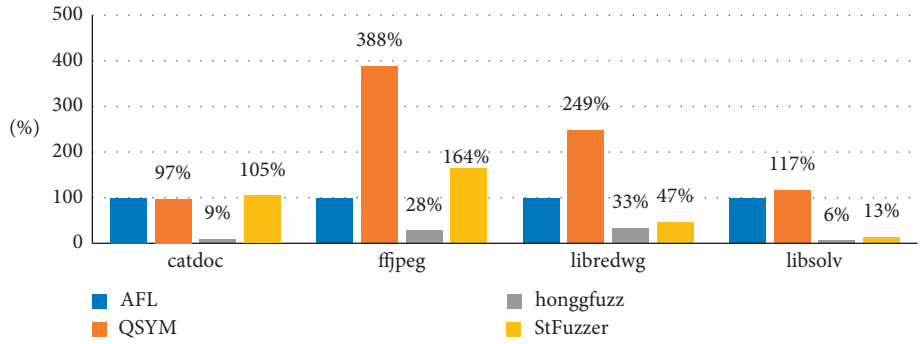

Figure 5: The average execution speed of each fuzzer.

StFuzzer has a better performance than AFL. In large-scale applications, the contribution-aware optimization introduces some performance overhead, resulting in a decrease in execution speed. However, the average execution speed of StFuzzer is still much faster than that of honggfuzz.

6. Conclusions

In this paper, we propose a contribution-aware optimization solution and implement a novel coverage-guided fuzzer for smart devices, named StFuzzer. We define the contribution of a basic block for the fuzzing process and describe that the basic block with higher contribution will be more useful to device exploration. We show an efficient algorithm to calculate the static contribution of a basic block based on control flow and the approach that evaluate the dynamic contribution through runtime information.

We evaluated StFuzzer with several state-of-the-art coverage-guided fuzzers in four real-world applications running on different smart devices. The result demonstrated that StFuzzer outperforms all baseline fuzzers in terms of vulnerability recognition, application exploration, and code coverage. And we also evaluated the performance overhead of the contribution-aware optimization, which showed that our optimization is lightweight and has good performance in both small-scale applications and large-scale applications.

Data Availability

The data used in this study are not available.

Conflicts of Interest

The authors declare no conflicts of interest.

Acknowledgments

This research was supported in part by the National Natural Science Foundation of China under grant U20B2046, Guangdong Province Key Research and Development Plan under grant 2019B010137004, Guangdong Higher Education Innovation Group 2020KCXTD007 and Guangzhou Higher Education Innovation Group 202032854, and Guangzhou University Graduate Student Innovation Ability Cultivation Funding Program (grant no. 2019GDJC-M16).

References

[1] M. Zalewski, "American fuzzy lop," 2013, http://lcamtuf.coredump.cx/afl/.

[2] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: application-aware evolutionary fuzzing," *National Down Syndrome Society*, vol. 17, pp. 1–14, 2017.

[3] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, and G. Grieco, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 861–875, San Diego, CA, USA, August 2014.

[4] M. Böhme, V. T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[5] O. Aflpin, "Github," 2018, https://github.com/mothran/aflpin.

[6] M. Eddington, *Peach Fuzzing Platform Whitepaper*, p. 34, 2011, https://www.peach.tech/wp-content/uploads/Peach-Fuzzer-Platform-Whitepaper.pdf.

[7] M. Böhme, V. T. Pham, M. D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, Dallas, TX, USA, November 2017.

[8] S. Gan, C. Zhang, X. Qin et al., "Collafl: path sensitive fuzzing," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, IEEE, San Fransico, CA, USA, May 2018.

[9] T. Yue, P. Wang, Y. Tang et al., "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)*, pp. 2307–2324, Boston, MA, USA, March 2020.

[10] P. Chen and H. Chen, "Angora: efficient fuzzing by principled search," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, IEEE, San Francisco, CA., USA, May 2018.

[11] H. Chen, Y. Xue, Y. Li et al., "Hawkeye: towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2095–2108, Toronto, Canada, October 2018.

[12] S. Wang, J. Nam, and L. Tan, "QTEP: quality-aware test case prioritization," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 523–534, Paderborn, Geramny, September 2017.

[13] W. You, X. Wang, S. Ma et al., "Profuzzer: on-the-fly input type probing for better zero-day vulnerability discovery," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, pp. 769–786, IEEE, Sanfransico, CA, USA, May 2019.

[14] S. Gan, C. Zhang, and P. Chen, "{GEYONE}: data flow sensitive fuzzing," in *Proceedings of the 29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 2577–2594, Boston, MA, USA, March 2020.

[15] Z. M. Jiang, J. J. Bai, K. Lu, and S. H. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *Proceedings of the 29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 2595–2612, Boston, MA, USA, March2020.

[16] U. Kargén and N. Shahmehri, "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*, pp. 782–792, Bergamo, Italy, August 2015.

[17] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, pp. 697–710, IEEE, SanFrancisco, CA, USA, May 2018.

[18] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proceedings of the nternational Static Analysis Symposium*, pp. 95–111, Springer, Veniue, Italy, September 2011.

[19] N. Stephens, J. Grosen, C. Salls et al., "Driller: augmenting fuzzing through selective symbolic execution," *National Down Syndrome Society*, vol. 16, pp. 1–16, 2016.

[20] I. Yun, S. Lee, M. Xu, Y. Zhang, and T. Kim, "{QSYM}: a practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th {USENIX} Security Symposium ({USENIX} Securiy 18)*, pp. 745–761, Baltimore, MD, USA, August 2018.

[21] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing," *National Down Syndrome Society*, San Diego, CA, USA, 2019.

[22] J. Peng, F. Li, B. Liu, K. Chen, and W. Huo, "1dvul: discovering 1-day vulnerabilities through binary patches," in *Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 605–616, IEEE, Portland, OR, USA, June 2019.

[23] Y. Chen, P. Li, J. Xu et al., "Savior: towards bug-driven hybrid testing," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1580–1596, IEEE, San Fransico, CA, USA, May 2020.

[24] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *Proceedings of the 30th {USENIX} Security Symposium ({USENIX} Security 21)*, Vancouver, Canada, August 2021.

[25] R. Swiecki, "Honggfuzz," 2016, http://code.google.com/p/honggfuzz.

[26] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 627–637, ACM, Paderborn, Germany, August 2017.

[27] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: machine learning for input fuzzing," in *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 50–59, IEEE, Urbana, IL, USA, November 2017.

[28] S. Y. Kim, S. Lee, I. Yun et al., "Cab-fuzz: practical concolic testing techniques for {COTS} operating systems," in *Proceedings of the 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pp. 689–701, Santa Clara, CA, USA, July 2017.

[29] H. S. Han and S. K. Cha, "Imf: inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2345–2358, Dallas, TX USA, November 2017.

[30] W. You, P. Zong, K. Chen et al., "Semfuzz: semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2139–2154, Dallas, TX USA, November 2017.

[31] Z. Tian, C. Luo, J. Qiu, X. Du, and M. Guizani, "A distributed deep learning system for web attack detection on edge devices," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 3, pp. 1963–1971, 2020.

[32] C. Luo, Z. Tan, G. Min, J. Gan, W. Shi, and Z. Tian, "A novel web attack detection system for internet of things via ensemble classification," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 8, pp. 5810–5818, 2021.

[33] M. Shafiq, Z. Tian, A. K. Bashir, X. Du, and M. Guizani, "CorrAUC: a malicious bot-IoT traffic detection method in IoT network using machine-learning techniques," *IEEE Internet of Things Journal*, vol. 8, no. 5, pp. 3242–3254, 2021.

[34] M. Shafiq, Z. Tian, Y. Sun, X. Du, and M. Guizani, "Selection of effective machine learning algorithm and Bot-IoT attacks traffic identification for internet of things in smart city," *Future Generation Computer Systems*, vol. 107, pp. 433–442, 2020.

[35] J. Liang, Z. Qin, S. Xiao, L. Ou, and X. Lin, "Efficient and secure decision tree classification for cloud-assisted online diagnosis services," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 4, pp. 1632–1644, 2021.