Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Adaptive scheduling-based fine-grained greybox fuzzing for cloud-native applications

Jiageng Yang[1], Chuanyi Liu[1*] and Binxing Fang[1]

Abstract

Coverage-guided fuzzing is one of the most popular approaches to detect bugs in programs. Existing work has shown that coverage metrics are a crucial factor in guiding fuzzing exploration of targets. A fine-grained coverage metric can help fuzzing to detect more bugs and trigger more execution states. Cloud-native applications that written by Golang play an important role in the modern computing paradigm. However, existing fuzzers for Golang still employ coarse-grained block coverage metrics, and there is no fuzzer specifically for cloud-native applications, which hinders the bug detection in cloud-native applications. Using fine-grained coverage metrics introduces more seeds and even leads to seed explosion, especially in large targets such as cloud-native applications.

Therefore, we employ an accurate edge coverage metric in fuzzer for Golang, which achieves finer test granularity and more accurate coverage information than block coverage metrics. To mitigate the seed explosion problem caused by fine-grained coverage metrics and large target sizes, we propose smart seed selection and adaptive task scheduling algorithms based on a variant of the classical adversarial multi-armed bandit (AMAB) algorithm. Extensive evaluation of our prototype on 16 targets in real-world cloud-native infrastructures shows that our approach detects 233% more bugs than go-fuzz, achieving an average coverage improvement of 100.7%. Our approach effectively mitigates seed explosion by reducing the number of seeds generated by 41% and introduces only 14% performance overhead.

Keywords Coverage-guided fuzzing, Cloud-native application, Fine-grained coverage metric, Scheduling algorithm, Exploration-exploitation problem

Introduction

Fuzzing is one of the most successful vulnerability detection techniques. Coverage-guided greybox fuzzing, a state-of-the-art category of fuzzing, is the most popular and effective approach to finding bugs in various software and hardware. For example, as a classic coverage-guided fuzzer, AFL [1] has found thousands of security bugs and has been optimized in various aspects by academic researchers. Most existing fuzzers focus on finding memory corruption bugs in targets written in low-level languages like C or C++. With the development of fuzzing, some solutions such as go-fuzz [2], python-afl [3], and RULF [4], pay attention to fuzzing in high-level languages.

Most coverage-guided greybox fuzzing can be modelled as a fuzzing loop that mutates a selected seed to generate new inputs, as shown in Fig. 1. First, the target application is analyzed and probes are instrumented to monitor code coverage. As the instrumented application is executed, the probes will modify the data structure (typically a fixed-size bitmap) that maps the execution space of the target to reflect the coverage. This mechanism is called the coverage metric. Then, the new inputs are executed as test cases in the instrumented application. Depending on the coverage metrics applied, the fuzzing loop stores interesting test cases as new seeds

*Correspondence:
Chuanyi Liu
liuchuanyi@hit.edu.cn
[1] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen 518055, Guangdong, People's Republic of China

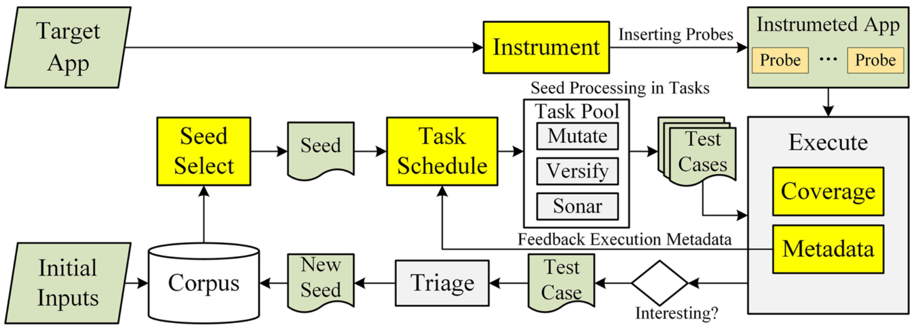


Fig. 1 The main fuzzing process of coverage-guided fuzzing. The components in yellow represent the focus of our solution. The green elements signify the inputs to the fuzzing process

in the corpus for future iterations. For coverage-guided fuzzing, coverage measurement is a critical metric for distinguishing outstanding test cases. Code coverage quantitatively measures the degree of testing for the target application, which is used to select seeds that trigger new execution states. The simplest coverage metric is block coverage. A typical block coverage metric maps all basic blocks to a fixed-size bitmap and records the hit of blocks by enabling the corresponding bit of the triggered block. However, it cannot track the order of blocks, especially when a block has multiple precedents, resulting in a loss of coverage information. Meanwhile, the fixed size bitmap may not be sufficient to represent all the blocks, especially for large applications. Unfortunately, existing fuzzers for Golang (such as go-fuzz and Go Fuzzing [5]) still use coarse-grained block coverage metrics, which hampers the effectiveness of fuzzing.

As the fuzzing loop iterates, new seeds are continuously added to the corpus, waiting to be scheduled for future mutations. The more sensitive the coverage metrics applied, the more frequently new seeds are added. In most fuzzers, a scheduled seed is mutated and executed in a fixed pattern, which determines the process capability of a fuzzer. Therefore, with a high-sensitive coverage metric, a sequential seed scheduler will be faced with too many new seeds to select the best seeds, i.e., the seed explosion problem.

A selected seed is mutated to generate inputs that can trigger new execution states. But simple random-based mutation leads to limited exploration of the test cases. Some work has employed adaptive mutation strategies to improve the efficiency of exploration [6–9]. For example, as shown in Fig. 1, go-fuzz uses three mutation strategies to help fuzzing break hard branches such as CRC32 checksum. Specifically, go-fuzz uses a mutation strategy called sonar to find associations between input bytes and unique branching behaviors, and uses a mutation strategy called versify to generate structural inputs. However, go-fuzz employs a fixed manner to arrange three mutation strategies for each seed, which is unsuitable for all seeds and thus limits the exploitation of the seeds.

Wang et al. [10] propose a concept of sensitivity to estimate the impact of a coverage metric on bug detection. The evaluation result of [10] shows that more sensitive coverage metrics do not always give better results, although a sensitive coverage metric distinguishes more states. A more sensitive coverage metric will generate more seeds to select, which may exceed the fuzzer's ability to schedule. Similarly, using more complex mutation strategies results in higher overhead and more seeds, which reduces the scheduling ability of the fuzzers. Especially, the larger the size of the target to be tested, the more seeds in the corpus, leading to a more serious seed explosion problem. Wang et al. [11] model the fuzzing as a multi-armed bandit problem and propose a hierarchical scheduler to solve the seed explosion caused by highly sensitive coverage metrics. Unfortunately, this work does not consider more smart mutation strategies to solve hard branch of targets.

Our observation is that cloud-native applications are much larger than the normal test targets written by Golang. Existing coverage metrics of fuzzers for Golang [2, 5] are simple block coverage metrics implemented with fixed size bitmaps, resulting in the coarse test granularity and inaccurate coverage information. Gan et al. [12] point out that inaccurate coverage results in loss of code coverage and even misses potential vulnerabilities. In this work, we propose an accurate edge coverage metric to optimize the test granularity of fuzzers for Golang. Specifically, we apply a hashing scheme similar to CollAFL [13], which considers the control flow information of all edges, and design an adaptive bitmap to capture accurate coverage information.

On the one hand, the accurate edge coverage metric leads to many more seeds and even seed explosion, especially for large targets such as cloud-native applications.

On the other hand, applying more complex mutation strategies further reduces the processing capability of the fuzzers, resulting in more severe seed explosion. In order to address the seed explosion problem, we regard the fuzzing loop as an adversarial multi-armed bandits problem [14], and propose smart seed and adaptive task schedulers to balance exploration and exploitation. More specifically, our smart seed scheduler selects a seed achieved high coverage to trigger more code region as exploitation and a fresh seed to discover surprising new branches as exploration. The adaptive task scheduler arranges flexible task strategies to mutate and execute seeds that generate new test cases tending towards exploration or exploitation.

To validate and evaluate our solution, we implemented a prototype, CloudFuzz based on go-fuzz. We performed a series of evaluation experiments on 16 targets from real-world cloud-native infrastructures. Compared to go-fuzz, CloudFuzz can find more bugs in real-world cloud-native applications. CloudFuzz achieved better coverage in all targets and improved coverage by more than 50% in most of targets. And CloudFuzz can achieve the same coverage faster than go-fuzz. Compared to go-fuzz with our accurate edge coverage metric, our seed and task schedulers significantly reduce the number of seeds in the corpus, mitigating the seed explosion problem caused by fine-grained coverage metrics.

Contributions. Our main contributions are:

- We propose and implement an accurate edge coverage metric that can achieve finer-grained and precise fuzzing in greybox fuzzers for Golang.
- We design smart seed selection and adaptive task scheduling algorithms based on a variant of the classical AMAB algorithm that balances exploration and exploitation for fuzzing, mitigating the seed explosion problem caused by highly sensitive coverage metrics.
- We implement our prototype CloudFuzz based on go-fuzz, which is suitable for bug detection in large cloud-native applications.
- We evaluate CloudFuzz on 16 targets from real-world cloud-native applications. The results show that CloudFuzz can detect more bugs and achieve higher code coverage, and effectively mitigate seed explosion in the process of fuzzing cloud-native applications. Compared to go-fuzz, CloudFuzz introduces only about 14% performance overhead.

Motivation

Coverage-guided fuzzing in cloud-native application

Fuzzing detects vulnerabilities in applications by feeding collected or generated random data into a target. Coverage-guided fuzzing is one of the most popular techniques for finding bugs. Coverage-guided fuzzing utilizes a heuristic scheduling algorithm to select interesting seeds for higher code coverage. Figure 1 illustrates the coverage-guided fuzzing process in more detail. Given a program and an initial corpus, the fuzzing process involves a sequence of fuzzing loops. In each fuzzing loop, the seed scheduler selects an interesting seed to mutate. The scheduled seed will be mutated to generate different test cases. The fuzzing process captures the coverage through instrumentation when a new test case is tested. If a test case triggers new coverage, it will serve as a seed for future rounds. However, there is a serious problem in the above fuzzing process, which is the seed explosion problem.

The seed explosion problem in fuzzing refers to a situation where the number of new seeds increases dramatically and uncontrollably. More specifically, a single mutation of a seed consists of multiple variation operators (such as bitflip operator, arithmetic operator, splicing operator, havoc operation and so on), which generates a large number of test cases. If the coverage metric is too fine-grained, a slight difference in execution will be measured as new coverage, resulting in most of these test cases triggering new coverage and being served as new seeds. In addition, if the size of the target application is too large, there will be too many execution states of the target, which will also result in many test cases being served as new seeds. These are the two potential causes of the seed explosion problem. Thus, a single mutation can add multiple seeds to the corpus, causing the number of new seeds to grow unrestrainedly, even exponentially, making it difficult to manage or prioritize seeds. In Fig. 2, to better illustrate the seed explosion problem, we show the growth curve of seeds in the corpus when go-fuzz (using a fine-grained coverage metric) tests cloud-native applications within 24 hours. In these targets, the number of new seeds continues to grow without a trend towards convergence.

There are many practical coverage-guided fuzzers for unsafe languages such as C or C++. The concept of fuzzing is useful for type-safe languages such as Golang. The fuzzers for C/C++ detect memory corruption vulnerabilities, such as heap overflow, Use-after-Free, and so on, which can be exploited to cause security problems such as code execution or privilege escalation. The fuzzers for Golang try to find bugs like out-of-bounds, null pointers and so on. Since Golang is memory-safe language, these bugs cannot lead to serious security issues, but can cause the unexpected behaviors/results of applications. In most cases, the fuzzer for Golang are used for unit testing. The fuzzing results demonstrate that go-fuzz is an effective tool for finding bugs in a single function or a group of

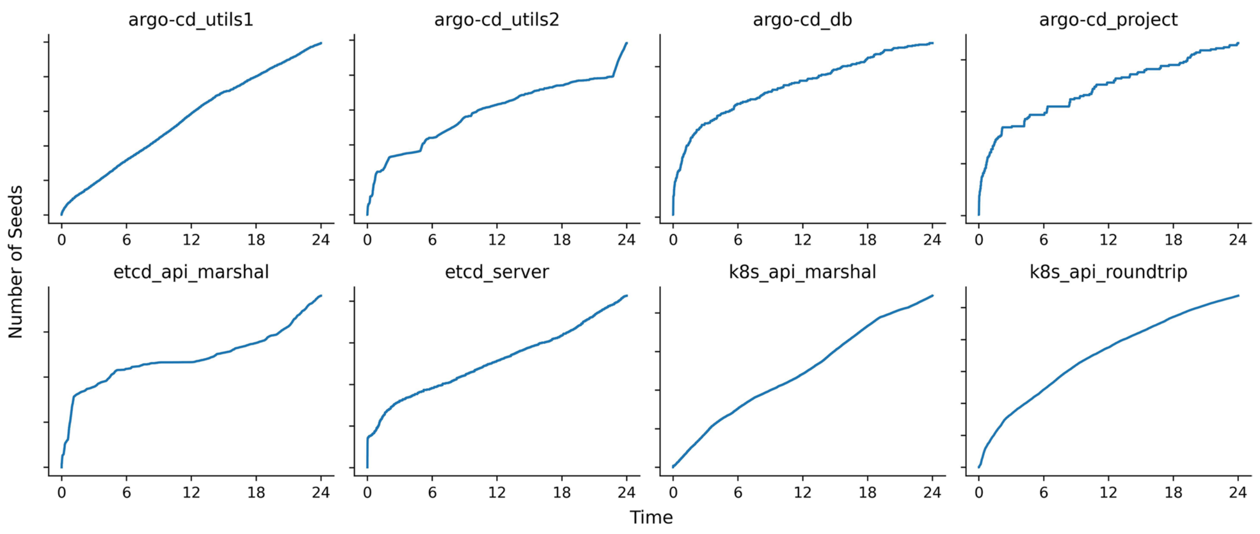


Fig. 2 The growth curve of the number of seeds in the corpus when fuzzing the real-world cloud-native application within 24 hours. Each subplot shows time on the horizontal axis and the number of seeds in the corpus on the vertical axis

functions in a unit. However, the situation is quite different for cloud-native applications. Although the majority of cloud-native applications are composed of type-safe languages like Golang. But the large size and multiple component interactions of cloud-native applications bring about new problems for most fuzzing techniques.

Firstly, most fuzzers for Golang use block coverage metrics, which is an appropriate metric for a test unit. But some research [10, 13] have shown that fine-grained coverage metrics like edge coverage metrics make fuzzers more sensitive and thus find more vulnerabilities. Second, the large size of cloud-native applications leads to severely inaccurate coverage measurement by coverage metric implementations, as demonstrated by [13]. Thirdly, numerous execution states frequently generate new seeds, exceeding the scheduling capabilities of fuzzers. These factors affect the effectiveness and efficiency of greybox fuzzing in cloud-native applications.

Coverage metrics for greybox fuzzing

As aforementioned, coverage-guided greybox fuzzing detects bugs or vulnerabilities via a feedback loop. The fuzzing process preserves test cases that trigger new coverage in the corpus and selects new seeds from the corpus to mutate and generate test cases for higher coverage. In the feedback loop, coverage is the most important indicator to measure the quality of test cases. There are many research works on new solutions for coverage metrics [11, 13, 15]. Go-fuzz [2] and VUzzer [16] utilize block coverage metrics to track whether new basic blocks are triggered. AFL-families [1, 17, 18] use edge coverage metrics to achieve more sensitivity. Wang et al. [10] proposed a formal definition of sensitivity to evaluate the impact of coverage metrics on the fuzzing process. Edge coverage metrics are more sensitive than block coverage metrics, contain sufficient coverage information, and introduce less overhead than more sensitive coverage metrics.

In fact, edge coverage metrics lead to finding more bugs than block coverage metrics, and finding bugs faster than block coverage metrics. In general, edge coverage metrics will increase the acceptable memory overhead and runtime overhead more than block coverage metrics. Edge coverage metrics strike a balance between efficiency and sensitivity [13]. Other more sensitive coverage metrics will cause more memory overhead and runtime overhead, but gain limited improvement in vulnerability detection. The previous research [10] shows that there is no grand slam coverage metric that can beat the others. In addition, we need to consider the more serious performance overhead caused by coverage metrics for fuzzing large cloud-native applications. Therefore, we apply an edge coverage metric to optimize go-fuzz for better vulnerability discovery capability.

Problem of coverage measurement

Coverage information is utilized to construct a fuzzing feedback loop. Inaccurate coverage information will reduces the effectiveness of the fuzzing feedback loop. Researchers have designed coverage metrics that take into account different critical objectives, such as memory access [19] or context information [20, 21]. Unfortunately, the implementation of coverage metrics is often limited. For instance, AFL applies a 64KB bitmap to capture all triggered edges during the fuzzing process. CollAFL [13] described the serious hash collision problem since the implementation of coverage measurement in AFL. AFL-sensitive [10] proposed several coverage metrics that considered multiple aspects of code and memory. However, the implementation of these coverage metrics still relies on different hash algorithms to represent the coverage in a fixed bitmap.

In popular fuzzers for Golang such as go-fuzz, the implementation of the coverage metric is similar to AFL. Go-fuzz uses a 64KB bitmap to calculate block coverage. As discussed above, the code coverage information measured by this approach is non-deterministic due to hash collisions. The larger the target size, the higher the hash collision rate in go-fuzz. Especially, for cloud-native applications, the size of most cloud-native applications far exceeds the coverage measurement capabilities of go-fuzz. As shown in Table 1, to accurately assess the hash collision problem in cloud-native applications, we calculated the hash collision rate of go-fuzz in typical cloud-native applications. The results show that if we fuzz the common targets in Table 1 using go-fuzz, the average of the hash collision rates in coverage measurement is 72.39%, which means that the measured coverage information is very inaccurate. The number of basic blocks for all targets is much larger than the bitmap size of go-fuzz. In kubelet internals components, the hash collision rate of the bitmap even exceeds 90%. The hash collision problem causes the accuracy of coverage to decrease to only one-tenth. Therefore, we employ an adaptive coverage measurement approach to avoid inaccurate coverage through the hash collision problem.

Scheduling strategy in greybox fuzzing

Fuzzers maintain a seed queue, scheduling interesting seeds one by one, and executing them to trigger bugs. The processing capability of a fuzzer is limited, so it is essential to prioritize some favored seeds to maximize code coverage. In other words, the fuzzer must schedule appropriate seeds to traverse as much code as possible in a limited amount of time. There are two factors that affect the scheduling ability of a fuzzer. First, the sensitivity of applied the coverage metric determines the number of new test cases generated. The more sensitive the coverage metric, the more test cases will be reserved in a round of the fuzzing loop. Secondly, the processing capability of a fuzzer determines how many test cases can be tested in a given time, i.e. the thoughput. The higher the thoughput, the higher the scheduling capability of a fuzzer. The above two factors will be more obvious for large fuzzed targets. Obviously, a larger fuzzed target will generate a larger seed queue and introduce more performance overhead. In cloud-native applications, this problem can significantly affect the scheduling capability of go-fuzz, as most cloud-native applications are too large to effectively select appropriate seeds for fuzzing.

To address the side effects of excessive seeds generated by the fuzzer, some work has attempted to model seed scheduling as an exploration and exploitation problem, and to utilize learning approaches to solve this problem. To be specific, the seed scheduler should select some fresh seeds to explore whether these new seeds could lead to crucial new coverage, and should also prioritize the exploration of some valuable seeds that have brought more new coverage in recent rounds than others. Wang et al. [11] modelled the fuzzing process as a multi-armed bandit problem and utilized a classical UCB1 algorithm to optimize the seed scheduler. The scheme regarded the generated seeds as arms and calculated the reward of each seed for seed scheduling. However, there are two problems that need attention in this scheme. First, the traditional MAB problem assumes a fixed number of arms, but the number of generated seeds increases as the fuzzer progresses. Second, the reward probability of each arm is fixed, which is different from the fuzzing process, where the probability of finding a new coverage decreases. EcoFuzz [22] proposed to

Table 1 Statistics of components in the common cloud-native applications

Application	Component	Basic blocks	Collision
argo-cd	util-db	473259	86.15%
argo-cd	project	530052	87.64%
containerd	config	88782	26.18%
etcd	api_marshal	166591	60.66%
etcd	etcdserver	159082	58.80%
kubernetes	apiextension	159919	59.02%
kubernetes	api_marshal	366502	82.12%
kubernetes	api_roundtrip	274873	76.16%
kubernetes	kubelet	741863	91.17%
kubernetes	kublet-server	509396	87.13%

Statistics includes the number of basic blocks and the corresponding collision ratio in go-fuzz's bitmap model the process of searching seeds and assigning energy as a variant of the Adversarial MAB (AMAB) problem. In the AMAB problem, it is assumed that the reward of each arm is arbitrary during each play, which is similar to the probability of discovering a new path during the fuzzer processing. SYZVEGAS [23] applied reinforcement learning to kernel fuzzing by modelling the fuzzing process as an AMAB problem. It proposed an automated way to identify the most promising task and invoked the best seed associated with the task. Overall, a sophisticated seed scheduler is essential for establishing the coverage-guided fuzzing feedback. It should balance the trade-off between fresh seed exploration and high coverage seed exploitation. Our contribution is to apply reinforcement learning to cloud-native applications for fuzzing processing and attempt to make use the AMAB model to schedule the most important tasks and the most promising seeds during the fuzzing feedback.

Design

We propose CloudFuzz, a dynamic fuzzing approach that applies a fine-grained coverage metric to improve the bug detection capacity of go-fuzz, which is suitable for cloud-native applications, by selecting appropriate seeds and scheduling optimal tasks.

Accurate fine-grained coverage metric

As aforementioned, coverage is one of the most important indicators to guide a fuzzer in exploring target applications and detecting bugs. Applying a proper coverage metric could help the fuzzing process to better understand the execution states of the fuzzed target. It is determined by appropriate coverage granularity and accurate coverage measurements.

Typical coverage solutions

The coverage metric a critical factor in coverage-guided fuzzing. A better coverage metric can improve the effectiveness of fuzzers. From the perspective of test granularity, there are three common types of coverage metrics, i.e., block coverage, edge coverage and path coverage. Many fuzzers use block coverage, such as VUzzer, libFuzzer, etc., because block coverage is the simplest way to measure. But, as mentioned above, block coverage loses the precedent information of a block. Edge coverage addresses this common situation, where a block has multiple precedents blocks. The edge coverage metric tracks the hit count of edges consisting of two adjacent blocks during program execution. Edge coverage is widely used by many fuzzers, such as the famous AFL. However, edge coverage cannot infer the execution order of each edge, which loses some of the coverage information. The path coverage metric tracks the order of edges, including the most complete coverage information. Another important factor in determining which coverage metrics to apply in our fuzzer is the performance overhead of measuring coverage. Although path coverage provides the most complete coverage information, but storing and capturing path coverage information will introduce an unacceptable overhead. Therefore, it is not a wise choice to apply path coverage to a common fuzzer. Edge coverage metrics achieve a trade-off between efficiency and coverage information. Many studies have shown that edge coverage metrics are a proven and reliable solution. Some work [10, 13] proposed the hash collision problem in the implementation of edge coverage in AFL. Furthermore, AFL utilizes a 64KB bitmap to represent all triggered edges during fuzzing. A triggered edge A→B is mapped into the bitmap by a simple hash formula as follows:

$$edge\_trans = (prev \ll 1) \oplus cur. \tag{1}$$

The *edge_trans* is used as the key to index into the bitmap to access the hit count of this edge. *prev* and *cur* are the key values of the blocks A and B that make up edge A→B. This oversimplified hash formula and the too small bitmap setting bring about a serious hash collision problem.

Go-fuzz uses a regular block coverage metric in go-fuzz. Similar to AFL, go-fuzz also employs a 64KB bitmap to capture all blocks and computes the hash for block A as follows:

$$block\_trans = sha1\{cur \mid cur \gg 8 \mid cur \gg 16 \mid cur \gg 24\}. \tag{2}$$

The *cur* is the key value of block A. Go-fuzz utilizes the sha1 algorithm to hash this buffer and translates the hashed data into a 32-bit unsigned integer that is used as the index of block A in the bitmap.

CloudFuzz's solution

Since an accurate edge coverage measurement will improve the effectiveness and efficiency of fuzzers, we argue that avoiding hash collisions in coverage tracking is a proper approach to address the fuzzing solution in cloud-native applications. In summary, we propose a solution that applies an accurate edge coverage metric to optimize the coverage tracking feature of go-fuzz and absolutely avoids inaccurate coverage due to compromised implementations. One observation is that the hash of an edge is determined by the current block and its precedents. Obviously, since an edge of a program is directed, if two edges end with different current blocks, they must be different edges. Moreover, if two edges end with the same current block, the necessary information to distinguish them is the difference between them, i.e., the precedents of the current block. Therefore, we discuss the edge coverage as two cases.

Algorithm 1 CloudFuzz's Coverage Solution

```
Require: Original Application
Ensure: Instrumented Application
 1: $(Single\mathbb{BB}, Multi\mathbb{BB}) :=$ GetCFG()
 2: $ES := \log_2 |bitmap| + 1$
 3: $Unsolv\mathbb{BB} := \varnothing$, $Solv\mathbb{BB} := \varnothing$
 4: for $b \in Multi\mathbb{BB}$ do
 5:     Paras = CalcParas(b)
 6:     if Paras = null then
 7:         $UnSolv\mathbb{BB} = UnSolv\mathbb{BB} \cup b$
 8:     else
 9:         $Solv\mathbb{BB} = Solv\mathbb{BB} \cup b$
10:     end if
11: end for
12: Instrument($Solv\mathbb{BB}$, $UnSolv\mathbb{BB}$, $Single\mathbb{BB}$, )
13: Function: CalcParas(b)
14: $flag$ = num(b.prev)
15: Selector $< x, y >$ in range $(0, ES)$
16: while w > 0 do
17:     $z$ = rand()%(b & $2^{\lceil \frac{ES}{2} \rceil}$)
18:     for each e $\in < b.prev, b >$ do
19:         $h$ = (e.prev $\gg$ x) $\wedge$ (b $\gg$ y) + z
20:         if bitmap[h] = 0 then
21:             flag = flag - 1
22:         end if
23:     end for
24:     w = w - 1
25:     if flag = 0 then
26:         return$< x, y, z >$
27:     end if
28: end while
29: return null
```

Case 1: Blocks with a single precedent. If a block has only one precedent, we could directly assign a identifier to this edge. Since the triggering the ending block of this edge belongs to only one situation, we do not need to distinguish which block is the precedent. We only need to find and assign a unique identifier to the block with a single precedent. To reduce the runtime overhead of coverage tracking, the unique identifier of blocks could be resolved as a constant at compile-time. In this case, the hit count of an edge can be recorded with only one array access operation, which greatly reduces the runtime overhead compared to AFL's hash calculation.

Case 2: Blocks with multiple precedents. If a block *A* has multiple precedents, we are unable to rely only on block *A* to distinguish edges ending up with block *A*. We have to compute the hashes of edges ending up with block *A* depending on block *A* and its execution precedent at runtime. If we use a fixed hash formula to compute the hashes of all edges, it is difficult to guarantee that we will definitely avoid the hash collision problem. Therefore, similar to [13], we utilize a dynamic hash formula to calculate the hashes of the edges as follows:

$$edge\_trans = (cur \ll x) \oplus (prev \ll y) + z. \quad (3)$$

where *(x, y, z)* are parameters to be determined, which may be different for different edges. The *(x, y)* parameters heuristically explore the bitmap to find unmapped space, then the *z* parameter is used to trim the proper bits for the block with multiple precedents. We use a heristical algorithm to traverse bitmap to find suitable parameters as Algorithm 1. If we cannot find the correct parameters for the edges within a certain window period to avoid a hash collision, we use the indexes of the free bits to assign identifiers to these edges. Specifically, we utilize a separate dictionary to record the identifiers (as the value) of these edges (as the key). In this case, coverage tracking involves two steps: searching for relevant identifier as an index and recording the hit count in the bitmap.

Overall. Algorithm 1 gives an overview of this solution. It starts by parsing the the Abstract Syntax Tree (AST) of the target application to extract the Control Flow Graph (CFG), assigning unique identifiers to each block, and counting the number of edges. This information is used to define the size of the bitmap and to distinguish between blocks with a single predecessor (SingleBB) or those with multiple predecessors (MultiBB). The exploration space required for coverage tracking is then determined (line 2). Next, the algorithm focuses on MultiBBs and computes appropriate hash parameters to facilitate efficient coverage tracking (lines 4∼11). We design a function to refine the hash parameters selection process (lines 13∼29). This function starts by extracting *(x, y)* parameters from the exploration space (line 15) and continues by constraining the z parameter within a given window based on the lower half-space of block b's identifier (line 17). It proceeds to assess all incoming edges to block b, ascertaining if the corresponding bits defined by the *(x, y, z)* tuple are unoccupied. If the bits are available, CalcParas returns the *(x, y, z)* tuple as the block's hash parameters and adds block b into the SolvBB set (line 9). In the event that suitable hash parameters are not found within the specified window, block b is placed into the UnsolvBB collection (line 7). Then, the algorithm seeks unused bits in the bitmap to track hit count of these blocks. It ensures that the remaining free bits are allocated to blocks within a single precedent. Finally, the algorithm inserts probes with varied parameters into each block type, culminating in the generation of the instrumented application. This process optimizes the allocation of bitmap space, ensuring that each basic block is monitored accurately for execution during testing.

It is worth noting that the goal of this solution is to analyzes and insert probes into the target application. Furthermore, it only requires to update the size of the bitmap by modifying the corresponding variable of the instrumentation library. This means that the fuzzer itself does not need to be recompiled. Once the edges of the target application are defined and the corresponding bitmap size is set, only the instrumented target application needs to be recompiled. This process ensures that the fuzzer remains unchanged, while the target application is adapted to reflect the new instrumented probes.

Gain and cost assessment

As mentioned above, we model the fuzzing process as an AMAB problem, where the seeds in the corpus are considered to be arms of bandits. In the AMAB problem, the costs and gains incurred in the round depend on which arm is chosen. On the one hand, during the fuzzing loop, when a seed is selected to participate in the next round, the results of various tasks based on this seed determine the gain of this seed selection. On the other hand, the execution time of these scheduled tasks implies a cost of this seed selection. In order to model and solve the AMAB problem of fuzzing, we need to reasonably estimate the costs and gains of a seed selection and assign a reward to each seed selected.

First, there is no doubt that coverage is the most important factor in determining the effectiveness of coverage-guided fuzzing. Thus, in CloudFuzz, we consider the new edge coverage(i.e., the number of new edges covered) triggered by a seed as a factor that affects round-specific gains. It is worth noting that vulnerabilities in real-world applications can be caused by a number of factors. These factors would help fuzzers to improve their ability to detect bugs. Our solution does not include these factors in the seed gain calculation, as we focus on solving the most fundamental and critical AMAB model to explore the approach to the optimal scheduling of fuzzing. Our solution can be easily be extended to cover these factors. Furthermore, the execution time of the seeds is the most critical factor to decide the efficiency of coverage-guided fuzzing. Similarly, we consider the execution time of a seed as the cost of it in a given round.

Gain estimation

Based on the above two factors, we estimate the gain of a seed in a given round. Let c_{s_i} be the number of new edges covered (i.e., the coverage improvement) by the seed s_i selected in the *ith* round, and t_{s_i} be the time cost in the *ith* round. Thus, the total coverage is denoted as $\mathcal{C} = \sum_{i=1}^{n} c_{s_i}$, and the total elapsed time as $\mathcal{T} = \sum_{i=1}^{n} t_{s_i}$. We can calculate the average energy efficiency ratio for discovering a new path as $\bar{e} = \frac{\mathcal{C}}{\mathcal{T}}$. According the parameters $\bar{e}$, we can estimate the expected coverage improvement in the *ith* round is $\bar{e} \cdot t_{s_i}$. In summary, we define the gain of the seed s_i in the *ith* round as:

$$g_i = c_{s_i} - \bar{e} \cdot t_{s_i}. \tag{4}$$

through the actual coverage improvement minus the expected coverage improvement. However, the average energy efficiency ratio $\bar{e}$ does not accurately describe the complexity of finding new paths in any one phase.

The reasons are as follows: 1) the number of total paths that can be executed of a program is finite; 2) the probability of a test case triggering a new path will decreases as the number of paths discovered increases. As a result, the energy efficiency ratios of finding a new path at different phases varies greatly. [22] proposed a relationship between the number of paths and the number of total executions in a typical fuzzing. During an actual fuzzing process, the trend in the number of paths found by a fuzzer is similar to the Curve S in Fig. 3. In the early phase ($0 \sim t_1$), the fuzzer quickly discovered a number of paths. Then, in the next phase ($t_1 \sim t_2$), the probability of a test case finding a new path was decreasing until a critical seed makes a breakthrough (at time t_2). The Curve S ($t_2 \sim t_4$) then repeats the trend of the above two stages until all paths are found or the coverage upper limit of the fuzzer is reached.

In order to precisely calculate the energy efficiency ratio at a given moment, we consider the Curve S as a function between total paths and total executions. Calculating the energy efficiency ratio at a moment t_i will translate into solving for the derivative of the function S at t_i. The energy efficiency ratio g_{t_i} at t_i is calculated as:

$$g_{t_i} = \lim_{t \to t_i} \frac{S(t) - S(t_i)}{t - t_i}. \tag{5}$$

Unfortunately, when fuzzing real-world programs, it is impossible to calculate the energy efficiency ratio at all moment precisely. Therefore, we use a compromising

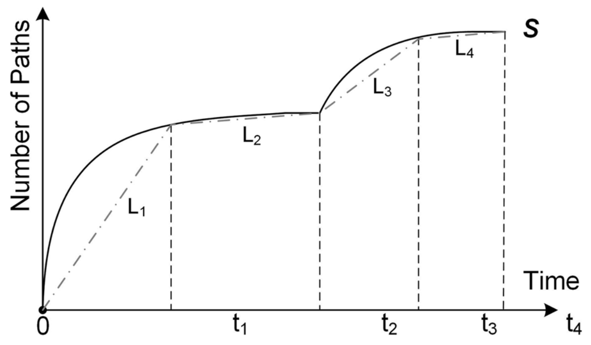


Fig. 3 A relationship between the number of paths and execution time during the fuzzing process

solution that defining a variable average energy efficiency ratio $\hat{e}$. The initial $\hat{e}$ is calculated as

$$\hat{e} = e_{(0,\Delta T)} = \frac{S(\Delta T) - S(0)}{\Delta T}. \tag{6}$$

Then, updating $\hat{e}$ with $\widehat{e} = e(n, n + k \cdot \Delta T)$ when the following conditions are satisfied:

$$\frac{|\hat{e} - e_{(n,n+k\cdot\Delta T)}|}{\hat{e}} \geq \pi. \tag{7}$$

where the current $\hat{e} = e_{(m,n)}$ and the parameters π could be configured to adjust the runtime overhead. Finally, we calculated the gain of seed s_i as:

$$g_i = c_{s_i} - \hat{e} \cdot t_{s_i}. \tag{8}$$

In the fuzzing process of go-fuzz, a seed could be used for triage, mutation, verifier and sonar tasks. Depending on task scheduling strategies, different seeds may go through different handling processes, but they must accomplish the four tasks listed above without exception. Based on the results of these tasks, we utilize Eq. 8 to estimate the rewards of the seeds over their life cycle.

Triage

First, a newly generated seed would be added to triage queue and await triage processing. The triage task contains two steps: minimization and smashing. The minimization step attempts to trim some bytes from seeds to reduce the size of the seeds while maintain the same coverage. Reducing the size of seeds saves some of the execution time in subsequent tasks. That is, in our model, minimization could cut off some of cost in subsequent executions, but there is no concern about a negative impact on the gain of seeds. Let the execution time of a seed s' (i.e., the cost of s') is $t^{s'}_{exec}$, then s' is minimized to a new seed s in one execution as $\Delta t^{s}_{exec} = t^{s'}_{exec} - t^{s}_{exec}$. Obviously, reducing the size of seed s' has benefits for all tasks that involve seed s', saving the cost of those tasks. All types tasks involve handling seeds with corresponding approach and executing them. For example, a mutation task mutates a seed firstly and then executes the generated test case. The smaller size of seeds also saves significant cost arising from handling. Cost savings in one handling of seeds are positively correlated with seed size reduction. It is not possible or necessary to accurately calculate the cost savings for each handling. We estimate the cost savings of seed s in one handling as $\Delta t^{s}_{exec} = \sigma(t^{s'}_{exec} - t^{s}_{exec})$, where $\sigma \in (0, +\infty)$ is a weight in order to indicate the correlation between handling cost and execution cost. Formally, we estimate the total cost savings resulting from minimization the seed s' as

$$\begin{aligned} e^{s}_{min}(m,n) &= m \cdot \Delta t^{s}_{hld} + n \cdot \Delta t^{s}_{exec} \\ &= (m \cdot \sigma + n)(t^{s'}_{exec} - t^{s}_{exec}). \end{aligned} \tag{9}$$

where m denotes the total times of handling seed s and n denotes the total times of executing seed s. In terms of calculating the gain of minimizing a seed S, we convert the cost savings into the desired coverage improvement using the parameters $\hat{e}$ and $e^{s}_{min}(m,n)$. In contrast, we should consider the cost savings of minimization as the gains of minimization. Let t^{s}_{min} be the cost of minimization seed s. Finally, we calculate the total gain from minimizing the seed s' after m times handling and n times execution as:

$$g^{s}_{min} = \hat{e} \cdot e^{s}_{min}(m,n) - \hat{e} \cdot t^{s}_{min}. \tag{10}$$

Another step in the triage phase is smashing. Smashing is a type of high-priority mutation that gives each new seed a minimal amount of attention. It is worth noting that smashing performs multiple deterministic mutating algorithms on a fresh seed, which is more efficient than absolute random mutating in the mutation phases. As a result, the gain of smashing is usually higher than the gain of mutation. Let $c^{s}_{smh}(i)$ be the gain (i.e., the number of new edges covered) of seed s in the *ith* variation of the smashing phase, t^{s}_{smh} be the total cost of smashing seed s. The total gain of smashing is the sum of the gains from each variation in the smashing phase. Formally, we calculate the total gain from smashing the seed s after n times variations as:

$$g^{s}_{smh} = \left(\sum_{i=1}^{n} c^{s}_{smh}(i)\right) - \hat{e} \cdot t^{s}_{smh}. \tag{11}$$

Finally, we calculate the total gain from triage for any one seed s as: $g^{s}_{tri} = g^{s}_{min} + g^{s}_{smh}$, which quantitatively reflects the coverage improvement and runtime overhead from triaging one seed s.

Mutation

Mutation is one of the most important and frequent tasks in the fuzzing process. According to the definition of fuzzing, a fuzzer needs to execute tested targets several times which relies on different random data generated by the mutation task. Furthermore, in go-fuzz, mutation tasks are performed nine times more than verification tasks, and one thousand times more often than sonar tasks. Through fresh test cases generated by mutating existing seeds, the fuzzing process will explore new execution states and trigger new vulnerabilities. Therefore, when a test case covers new edges, it indicates that a mutation task has produced a valuable test case, which serves as an important factor in measuring the gain of mutation tasks. More specifically, a mutation task consists of two operations: first, mutating a scheduled seed from the corpus to generate a new test case; second, executing the new test case in the target application, measuring coverage information and whether a crash occurs.

Let $t^S_{mut^1}(i)$ be the cost of step 1 in *ith* round of mutation tasks on seed *s*, and $t^S_{mut^2}(i)$ be the cost of step 2 in *ith* round of mutation tasks on seed *s*. The cost of a mutation task in *ith* round $t^S_{mut}(i) = t^S_{mut^1}(i) + t^S_{mut^2}(i)$. In theory, what really makes a difference to the coverage improvement is the mutation operation on the seeds (the first step). Executing test cases generated by mutations is used to check whether the test cases have improved coverage. In essence, the execution operation (step 2) does not improve coverage. Let $c^s_{mut}(i)$ be the coverage improvement of *ith* mutating the seed *s*, in a narrow sense, the gain of *ith* mutating the seed *s* can be calculated as:

$$g^s_{mut} = c^S_{mut}(i) - \hat{e} \cdot t^S_{mut^1}(i). \tag{12}$$

In this formula, we only consider the cost of the true mutation operation and do not include the execution operation, which only serves as a check. In this approach, our policy favours mutating seeds to explore the target application more often. Otherwise, in a generalized sense, we consider the execution operation in a mutation task to define the gain of *ith* mutating the seed *s* as:

$$g^s_{mut} = c^S_{mut}(i) - \hat{e} \cdot t^S_{mut}(i). \tag{13}$$

In this approach, our policy incorporates the expense of executions due to mutate a seed, which focuses on a balance of exploration and exploitation. In summary, we calculate the total gain from mutation tasks when n mutations are made to seed *s* as:

$$g^s_{mut} = \sum_{i=1}^{n} c^s_{mut}(i) - \hat{e} \cdot t^S(i). \tag{14}$$

where $t^S(i)$ is decided on our policy, which could be $t^S_{mut^1}(i)$ (based on formula 12) or $t^S_{mut}(i)$ (based on formula 13).

Versify and sonar

Versify and sonar serve a similar purpose and apply analogous ideas to achieve their goals. The versify task uses a heuristic algorithm to recognize internal structures of seeds in the corpus and generate new test cases containing similar structures. The experimental results shows that versify task could help go-fuzz find more seeds and cover more basic blocks. Unlike the mutation task, the versify task only performs a series of continuous variant operations on the current seed. This allows us to estimate the gain of a versify task as a whole. Let c^S_{ver} be coverage improvement of seed S in a versify task, and t^S_{ver} be the total cost of versifying seed S. We calculate the grow of versifying seed S as: $g^s_{ver} = c^s_{ver} - \hat{e} \cdot t^S_{ver}$.

Similar to the versify task, a sonar task includes a series of constant variant operations. A sonar task consists of two phases of operations. The sonar task identifies crucial comparison statements where both operands are dynamic and marks these statements as sonar sites at complier-time, replacing the left value of a sonar site with its right value at runtime. Fortunately, most of the cost of a versify task is caused by its first phase, namely the cost in complier-time, which has no impact on the runtime overhead. Thus, we do not need to take into account the overhead of the first phase when calculating gains. Let c^S_{sor} be coverage improvement of seed S in a sonar task, and $t^S_{sor_2}$ be the runtime cost of the second phase in the corresponding sonar task. We calculate the grow of sonar a seed S as: $g^s_{sor} = c^S_{sor} - \hat{e} \cdot t^S_{sor_2}$.

Smart seed selection policy

Existing seed selection policies

Seed selection is a crucial step in determining which seeds will be prioritized for the next rounds. Existing studies have shown that, a good seed selection policy could improve the efficiency of the fuzzer and find more bugs in finite fuzzing rounds. There are two major categories of seed selection policies: constant feature policies and feedback-based trade-off policies.

First, constant feature policies cause the fuzzer to preferentially select seeds according to some runtime features that are of interest to the established policies. For instance, VUzzer [16] believed that deeper basic blocks are more difficult to be triggered and prioritized seeds which traversed deeper blocks. FIFUZZ [24] proposed a context-sensitive SFI-based approach to guide fuzzing exploring error handling code. Policy in this category typically direct fuzzers to exercise paths with fixed features that are considered prone to producing bugs or vulnerabilities. Related works have theoretically demonstrated that these proposed features do correlate to some degree with the occurrence of bugs, and have shown experimentally a positive correlation with bugs. However, these policies applied to the fuzzers limit the accessible, interesting regions of the target application. The fuzzers focus on exploring only the specified regions that are defined in the applied policies, but igonre valuable regions that are not defined in the applied policies, reducing the fuzzers' ability to find bugs.

Second, feedback-based trade-off policies make the fuzzer to select a few high-quality seeds that have exercised significantly more new coverage in historical rounds as exploitation, and to select fresh seeds that have rarely been selected but may lead to surprising new coverage as exploration. The most important objective of policies in this category is to strike a balance between exploitation and exploration. For instance, AFLFast [18] attempted to assign different energy to different seeds based on the frequency at which the seeds were chosen and the distribution density of the paths exercised by the seeds, thus avoiding excessive exercise of high-freqnency paths. EcoFuzz [22] proposed a scheme that models seed selection as a variant of adversarial MAB problem, which assigns different energies to seeds by estimating their reward probabilities, assigning the appropriate energy to each arm to achieve a trade-off between exploitation and exploration. Similarly, SYZVEGAS [23] applied an Exp3-IX-like algorithm to solve the AMAB problem of seed selection. Unlike constant feature policies, feedback-based trade-off policies not only explot seeds with expected features, but also explore fresh seeds to achieve unintended but valuable new coverage. Recent work has shown that feedback-based trade-off policies utilizing learning-based algorithms to address seed selection, modelled as an exploitation and exploration problem, are effective and efficient, improving the speed of exercise applications and bug detection.

Algorithm 2 Seed Selection Algorithm

```
Require: Corpus $\mathbb{S}$
Ensure: Scheduled seed $s_i$ in the $ith$ round
 1: Set $|\mathbb{S}| = K$
 2: $\gamma = \sqrt{\frac{3k \log k}{2T \cdot (e-1)}}$
 3: t = 0
 4: for each seed $i \in \mathbb{S}$ do
 5:     $\hat{G}_i(0) = 0$
 6:     $w_i(0) = 1$
 7: end for
 8: while t < T do
 9:     $p_i(t) = (1 - \gamma) \cdot \frac{w_i(t)}{\sum_{j=1}^{k} w_j(t)} + \frac{\gamma}{k}$
10:     $i_t \leftarrow$ SelectSeed($\mathbb{S}, t$) according to $p_i(t)$
11:     $G_{i_t}(t) \leftarrow$ CalcGain($i_t$)
12:     for each seed $i \in \mathbb{S}$ do
13:         if i == $i_t$ then
14:             $\hat{G}_i(t) = \frac{G_i}{p_i(t)}$
15:         else
16:             $\hat{G}_i(t) = 0$
17:         end if
18:     end for
19:     for each seed $j \in \mathbb{S}$ do
20:         $w_j(t+1) = w_j(t) \cdot e^{\frac{\gamma \cdot \hat{G}_j}{K}}$
21:     end for
22:     t = t + 1
23: end while
```

CloudFuzz's seed selection policy

As aforementioned, feedback-based seed selection policies can obviously improve fuzzer's ability to detect bugs. Based on this observation, we construct a feedback mechanism between the historical gain of the selected seeds and the preference of the selection strategies, which helps CloudFuzz to select proper seeds to strike a balance of exploration and exploitation. Similar to existing work [22, 23], we model the seed selection process as an Adversarial MAB (AMAB) problem. Each seed in the corpus is regarded as an arm of bandits, and the goal of seed selection is to obtain the maximum of the sum of the gains from all the selected seeds. Fortunately, the Exp3 algorithm is a proven solution for complex AMAB problems. In this work, we have adapted the typical Exp3 algorithm to suit our gain assessment method, as shown in Algorithm 2.

The Exp3 algorithm assigns a weight to each arm in every round (line 6). The weights determine which arm will be selected for next round (line 10). Then, the weights will change depending on the reward earned in each game (line 20). In a word, we have implemented a feedback loop between the choices of arms and the rewards of arms. The core step of seed selection algorithm is to draw a seed i_t to participate the next round according to the distribution $\{p_1(t), p_2(t), ..., p_i(t)\}$ that is the distribution of estimated reward probabilities for each seed in round t (line 10). Then, we calculate the total reward of the selected seed i_t from achieved all tasks in round t (line 11). The CalcGain function in Algorithm 2 consists of two steps: 1. Calculate cumulatively the total reward of seed i_t in round t; 2. Normalize the total reward of seed i_t. Following formulas in the previous section, we calculate the total reward $g^{i_t}(t)$ of a seed i_t through accumulating the gains of all tasks (including mutation, versify and sonar, in the case of fresh seed t_i, a triage task) in round t. The total reward $g^{i_t}(t)$ can take values from $(-\infty, +\infty)$. However, the Exp3 algorithm requires the reward of each arm to take values in the range [0, 1]. So, the second step of CalcGain function is normalizing the total reward $g^{i_t}(t)$ to [0, 1]. We apply a common logistic function to normalize the total reward as follows:

$$G_{i_t}(t) = \frac{1}{1 + e^{-g^{i_t}(t)}}. \tag{15}$$

In this step, we excluded the most common Z-score normalization or max-min normalization, because these methods require maintenance and traversal of the historical gain data, which can cause a significant performance overhead. In addition, we calculate the estimated reward

$\hat{G}_i(t)$ for each seed i based on its reward and reward probability (line 12∼16). The estimated reward of each seed will affect the weight of each seed in the next round (line 20), which achieves the feedback mechanism.

Adaptive task scheduling strategy

As mentioned earlier, go-fuzz contains four tasks for different goals: triage, mutation, versify and sonar. In go-fuzz, it assigns different weights to the seeds in the corpus to select the appropriate seeds for the next round fuzzing. However, there is no scheduling algorithm that can judge which tasks should be given priority. Some work has pointed out that allocating different amounts of energy to the task being performed based on the coverage features of the seeds could result in significant energy savings.

Similar to seed selection, the task scheduler can be modelled as a series of exploitation and exploration behaviours. For exploitation, we should prioritize tasks that generate more high-value seeds, which will promise the fuzzing process attains a higher coverage in a short period of time. For exploration, we should also attempt to make use of high-cost but potentially effective tasks that may help the fuzzing process break through bottlenecks that are being experienced (e.g., hard-to-guess branches, etc.). Invoking different tasks for the same seed will result in the seed triggering diverse coverage and execution results. We are concerned with a trade-off task scheduling strategy that accomplishes as high coverage as possible with finite resources and time.

In order to ensure the robustness of CloudFuzz, we reuse the exploitation-exploration strategies decision unit to reduce the redundant and complexity of CloudFuzz. More specifically, we also utilize the Exp3 algorithm to achieve the adaptive task scheduling strategy. The Exp3 algorithm is a classic and useful approach of addressing the trade-off between exploitation and exploration. Compared to the Exp3 algorithm in the seed selection, there are two differences with the Exp3-like algorithm in the task scheduling. First, the number of arms in the task scheduling scenario is a constant as there are only four fixed tasks. But in the case of the seed selection, since the number of seeds is increasing, the number of arms is also escalating. Second, the calculation of rewards is different for each arm when it is selected. In the seed selection scenario, the rewards of seeds are assigned as the sum of reward from all tasks in current round. But in the task scheduling scenario, the reward of the task fits perfectly with our novel gain assessment approach. Our gain assessment is built on the increased coverage and time cost per task.

Implementation

We implemented CloudFuzz on top of go-fuzz, as go-fuzz still one of the most popular fuzzers for Golang. Our implementation consists of approximately 2,000 lines of code. It is worth noting that, our coverage metric solution and AMAB-based exploration &exploitation strategies could be applied to any coverage-guided fuzzer for Golang. Existing work shows that these approaches could improve the efficiency of fuzzers for the C language.

As aforementioned, at compiler-time, our implementation obtains the number of nodes in ASTs (Abstract Syntax Tree) of fuzzed targets and traverses these nodes to build appropriate instrumented functions for different coverage cases, as shown in Accurate fine-grained coverage metric section. It is different from the fixed size bitmap in go-fuzz's implementation, we make use of a dynamically sized slice as the coverage bitmap to adapt to our coverage metric. In addition, we rebuilt the instrumented runtime library using Golang assembly, which improves the execution efficiency of CloudFuzz.

We implemented an Exp3-like algorithm to achieve an AMAB-based seed slection policy and task scheduling strategy. As shown in Gain and cost assessment section when a seed T is tested, we will estimate its gains by counting the coverage improvement and runtime overhead caused by seed T. In order to achieve fine-grained task scheduling, we will also collect information on the coverage increasing and time spent from seed T at different stages of the fuzzing process. Eventually, our Exp3-like algorithm allows CloudFuzz to balance exploration and exploitation behaviors through the smart seed selection policy and the adaptive task scheduling strategy.

Evaluation

In this section, we evaluate our prototype CloudFuzz, to validate whether our improvement have boosted fuzzing performance. Considering with our improvement solutions, we conducted a series of experiments in CloudFuzz with different configurations on various cloud-native applications to answer following questions.

- **RQ1**: Can CloudFuzz achieve higher coverage than the baseline on cloud-native applications?
- **RQ2**: How much overhead of an impact does our improvement compared to the baseline?
- **RQ3**: Can CloudFuzz detect more bugs than the baseline on cloud-native applications?
- **RQ4**: How effective is our smart seed selection policy in solving the state explosion problem in cloud-native application?
- **RQ5**: Can our task scheduling strategy improve the fuzzing performance compared to the baseline?

Experiments setup. We fuzzed each target for 24 hours (on two cores) and repeated each experiment 10 times to reduce the effects of randomness. All evaluation experiments were done on 64-bit machine with 24-core, 48-thread CPU@2.1GHz, 64GB of RAM, and Ubuntu 20.04 as server OS. The version of Golang is 1.18.

Dataset. Our dataset includes 16 real-world cloud-native utilities. These utilities were selected from cncf-fuzzing test suits based on the importance of the cloud-native infrastructure, popularity in the community and diversity of categories. See Table 2 for details of the dataset.

Fuzzers. We selected four different fuzzers for our experimental evaluation. These include the baseline fuzzer and variants that belong to our CloudFuzz framework. The CloudFuzz variants enable different optimizations, which separately and intuitively evaluate the impact of the proposed optimizations on the testing process described in this paper. See Table 3 for details of the fuzzers in the following evaluation.

Code coverage in CloudFuzz

Results from the analysis of experiments results on 16 real-world cloud-native applications demonstrates that CloudFuzz outperforms existing fuzzers in terms of code coverage. In particular, CloudFuzz consistently overcomes coverage bottlenecks that often hinder the baseline fuzzer, allowing for more comprehensive code exploration. In addition, CloudFuzz quickly reaches the coverage upper limit of the baseline compared to the baseline fuzzer.

Code coverage is one of the important metrics to evaluate the effectiveness of the coverage-guided fuzzer. We ran CloudFuzz and go-fuzz on 16 cloud-native applications for 24 hours, and compared their coverage metrics and path exploration capabilities.

Table 2 The dataset from cncf-fuzzing

Number	Application	Version	Component	Function
No 1	argo-cd	2.5.11	db	CreateRepoCertificate
No 2	argo-cd	2.5.11	diff	StateDiff
No 3	argo-cd	2.5.11	normalizer	Normalize
No 4	argo-cd	2.5.11	server	CreateToken
No 5	argo-cd	2.5.11	server	ValidateProject
No 6	containerd	1.6.18	docker	ParseHostsFile
No 7	etcd	3.5.7	api	api_marshal
No 8	etcd	3.5.7	server	V3Server
No 9	istio	1.17.0	mesh	ParseMeshNetworks
No 10	istio	1.17.0	mesh	ValidateMeshConfig
No 11	kubernetes	1.24.10	apiextension	ConvertToTable
No 12	kubernetes	1.24.10	api	api_marshal
No 13	kubernetes	1.24.10	api	api_roundtrip
No 14	kubernetes	1.24.10	kubelet	HandlePodCleanups
No 15	kubernetes	1.24.10	kubelet	Server
No 16	kubernetes	1.24.10	kubelet	SyncPod

This table contains the tested functions as the targets in the last column, the corresponding components and version information in the cloud-native applications where these targets are located. To facilitate the description of the evaluation results, we assign a number to each target, as shown in the first column

Total coverage improvement

Table 4 shows the mean maximum coverage achieved by different fuzzers within 24 hours on targets of Table 2. The results of Table 4 show that CloudFuzz significantly improve the maximum coverage achieved by fuzzers on real-world targets compared to go-fuzz. In the last column of Table 4, it indicates the coverage increase rate of CloudFuzz compared to go-fuzz. Experimental results show that compared to go-fuzz, CloudFuzz achieved an average coverage improvement of 100.7% on the 16 targets, meaning that CloudFuzz outperformed go-fuzz. In addition, CloudFuzz achieved at least 50% higher code coverage on most of the 16 real-world targets. With the exception of the No 10 target, where CloudFuzz achieved a 5.16% improvement in code coverage, the remaining targets showed significant coverage gains. For all other targets, CloudFuzz delivered coverage improvements in excess of 30%. Impressively, for more than a third of the targets, CloudFuzz achieved maximum coverage that were over 100% higher than the baseline.

Furthermore, the third column illustrates the maximum coverage achieved by go-fuzz-well. Go-fuzz-well employed the identical seed selection algorithm and task scheduling strategy as go-fuzz. Thus, a comparison between go-fuzz and go-fuzz-well can see the impact of the coverage metric that employs an adaptive bitmap.

The results demonstrate that the use of our coverage metric in go-fuzz-well leads to significant improvements in code coverage. On average, go-fuzz-well consistently outperforms go-fuzz across most targets. For instance, in target No 2, go-fuzz-well achieves a coverage improvement of 33.08% over go-fuzz. Notably, target No. 14 demonstrates a remarkable increase in coverage of 46.68%, indicating that our coverage metrics can yield substantial enhancements in specific scenarios. In the majority of targets (10 out of 16), go-fuzz-well achieves more than a 10% increase in coverage.

This analysis highlights the importance of adopting advanced coverage metrics in fuzzing tools to achieve superior results. Moreover, when compared to go-fuzz-well, CloudFuzz and CF-seed exhibit even more significant coverage improvements. This indicates that the smart seed selection and adaptive task scheduling

Table 3 The fuzzers used in the evaluation experiments and the optimizations they enabled

Fuzzer	**Optimizations**			**Description**
	Cov	**Seed**	**Task**	
go-fuzz				Using go-fuzz as the baseline fuzzer because go-fuzz is a common fuzzer for Golang and we implemented CloudFuzz by enabling optimizations on top of go-fuzz.
go-fuzz-well	✓			A variant of go-fuzz that enables our coverage solution to observe the influence of our fine-grained coverage metric.
CF-seed	✓	✓		A variant of CloudFuzz that enhances seed selection optimization based on go-fuzz-well to evaluate the effectiveness of our seed selection strategy.
CloudFuzz	✓	✓	✓	The prototype of our solution that enables all optimizations proposed in our paper to evaluate the effectiveness and efficiency of our solution. Especially, compared with CF-seed to evaluate the impact of the task scheduling optimization.

This table presents the selected fuzzers for evaluation in the first column, the corresponding optimizations in the middle column, the goals and the reasons for their selection in the last column. The "Optimizations" column includes three optimizations proposed in this paper: fine-grained coverage metric (denoted as Cov), smart seed selection (denoted as Seed), and adaptive task scheduling (denoted as Task)

Table 4 The mean of the maximum coverage in evaluation experiments

Num	**go-fuzz**	**go-fuzz-well**	**Cov Inc go-fuzz-well**	**CF-seed**	**Cov Inc CF-seed**	**Cloud-Fuzz**	**Cov Inc CloudFuzz**
No 1	2015	2341	16.18%	2676	32.80%	4318	114.29%
No 2	1578	2100	33.08%	5950	277.06%	5658	258.56%
No 3	4439	4984	12.28%	6381	43.75%	6877	54.92%
No 4	3391	3728	9.94%	7551	122.68%	8421	148.33%
No 5	503	665	32.21%	991	97.02%	1073	113.32%
No 6	1296	1411	8.87%	1743	34.49%	1924	48.46%
No 7	15541	19025	22.42%	24559	58.03%	25932	66.86%
No 8	3905	4117	5.43%	4869	24.69%	11509	194.72%
No 9	3573	4075	14.05%	4741	32.69%	6957	94.71%
No 10	6317	6317	0.00%	6316	-0.02%	6643	5.16%
No 11	2890	3011	4.19%	3118	7.89%	4135	43.08%
No 12	40210	45800	13.90%	52694	31.05%	66631	65.71%
No 13	36729	47862	30.31%	69555	89.37%	70762	92.66%
No 14	1823	2674	46.68%	3468	90.24%	3671	101.37%
No 15	5262	5418	2.96%	5683	8.00%	7249	37.76%
No 16	1473	2203	49.56%	4289	191.17%	4003	171.76%

The maximum coverage achieved by CloudFuzz, CF-seed, go-fuzz-well, and go-fuzz on the targets listed in Table 2. The coverage increase rates for CloudFuzz, CF-seed and go-fuzz-well over go-fuzz are denoted as Cov Inc CloudFuzz, Cov Inc CF-seed and Cov Inc go-fuzz-well, respectively

proposed in this study result in more effective coverage enhancements.

In total, empirical evidence shows that CloudFuzz outperforms go-fuzz in terms of state exploration. Our solution is specifically designed to excel in testing scenarios within cloud-native applications, enabling CloudFuzz to uncover a greater number of execution states, thereby enhancing its effectiveness in detecting potential bugs.

Coverage growth over time

Figure 4 shows the coverage growth of various fuzzers within 24 hours on 16 targets from Table 2. It reveals an interesting fact via the coverage growth curve. On the one hand, we can observe that the coverage of CloudFuzz grows faster than go-fuzz in the early stages of fuzzing. Figure 4 shows how quickly CloudFuzz reaches the upper coverage limit of the baseline fuzzer in 24 hours. We can see that, with the exception of two targets (No 3, No 10), CloudFuzz achieved the same coverage in 1 hour as go-fuzz did in 24 hours for all the remaining targets in Table 2. For target No 10, CloudFuzz did not open a significant gap with go-fuzz in terms of coverage growth. Overall, CloudFuzz exhibits faster coverage growth than go-fuzz, and even in the short term, CloudFuzz triggers a

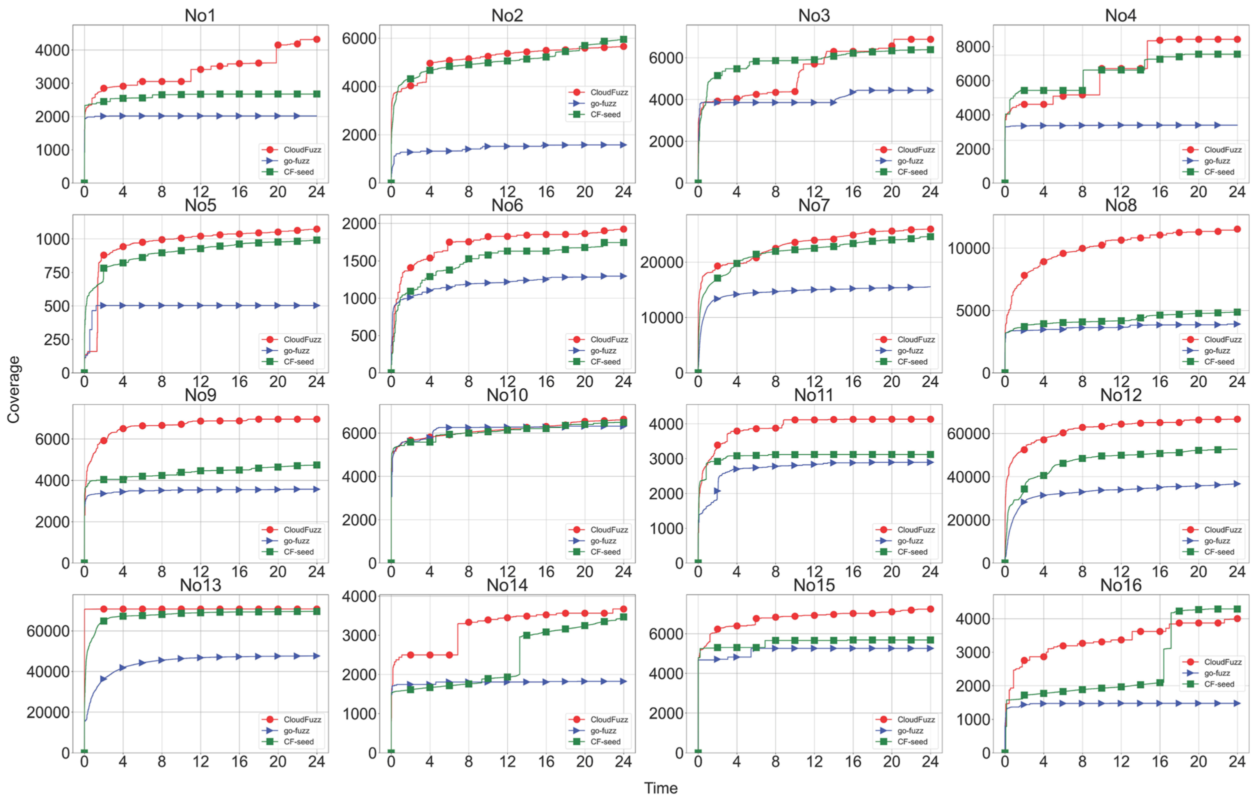


Fig. 4 The average coverage growth on targets from Table 2 discovered by each fuzzer over 24 hours

greater number of execution states than go-fuzz triggers over a longer period of time, as our scheduling strategy tends to apply exploitation at an early stage to achieve higher coverage quickly.

On the other hand, we can observe that for most targets go-fuzz discovered more new coverage in the early stages, but go-fuzz's coverage growth rate decreased over time. Eventually, go-fuzz's coverage growth rate converges to zero, i.e., go-fuzz has reached its coverage bottleneck and can no longer find new coverage. Instead, it is evident that in the 75% of the targets CloudFuzz has been able to break through the previous bottlenecks and has explore more new coverage. For instance, in target No 16, we observe that go-fuzz was unable to trigger the true working logic of the target's code, as evidenced by the low coverage of go-fuzz, whereas CloudFuzz frequently triggered new coverage (i.e., new code regions) in all phases of fuzzing, suggesting that CloudFuzz's exploration strategies were effective. For example, in target No 4, we can see that CloudFuzz reached the first bottleneck at around 2 hours into the run, which CloudFuzz broke it after 3 hours of exploration, and then broke the bottlenecks twice in a row. Eventually, CloudFuzz achieved an improvement in coverage of almost 100% compared to the coverage of the first bottle.

CloudFuzz's coverage growth curves reflect the balance of exploitation and exploration attained by our solution, demonstrating the effectiveness and efficiency of our AMAB-based seed selection and task scheduling solution. In the early stages of fuzzing, CloudFuzz tends to exploit interesting seeds to trigger as many code regions as possible. When it encounters a puzzle that has prevented the fuzzer from finding new coverage, CloudFuzz turns back to exploration, looking for fresh seeds to cover new features.

Bug detection in CloudFuzz

In order to address RQ1, we conducted a series of experiments using real-world cloud-native applications. The experiments conducted on our dataset 2 demonstrated that CloudFuzz outperformed the baseline by identifying a greater number of unique crashes. In particular, CloudFuzz successfully crashed all target applications, whereas the baseline fuzzer fail to detect crashes in most of the evaluated targets.

In these experiments, we evaluate the ability of fuzzers to detect unknown bugs in real-world applications. Table 5 shows the number of crashes in cloud-native applications observed over ten iterations of the experiments. It is important to emphasize that the targets

Table 5 Number of unique crashes detected by go-fuzz and CloudFuzz

Application	go-fuzz	CloudFuzz
argo-cd	0	124
containerd	0	24
etcd	6	20(233%)
istio	0	12
kubernetes	55	566(929%)

tested in our experiments represent the latest versions of the respective products. These cloud-native applications have been extensively tested by the vendors, either through the CNCF-fuzzing project or by using the Golang's fuzzing functionality. On average, go-fuzz only detects crashes in two targets, while CloudFuzz successfully crashes all targets. In the two targets that are crashed by go-fuzz, the number of crashes triggered by CloudFuzz significantly surpasses that of go-fuzz. Specifically, in targets of Kubernetes, CloudFuzz found nearly ten times as many unique crashes compared to go-fuzz. In targets of etcd, CloudFuzz detected 233% more unique crashes than go-fuzz. The unique crashes identified by go-fuzz in these two specified targets were also observed within the unique crashes triggered by CloudFuzz. In the remaining targets, CloudFuzz detected 160 unique crashes, while go-fuzz found no bugs in these targets.

It should be emphasized that the programs in which these targets reside perform a critical infrastructure functions within the cloud-native ecosystem, such as kubernetes and etcd, among others. Finally, we analyzed these unique crashes, and found that CloudFuzz found 10 bugs in real-world targets and go-fuzz only found 3 bugs in same targets. CloudFuzz detected 233% more bugs than go-fuzz. The bugs identified by CloudFuzz have the potential to precipitate Denial of Service attacks on cloud-native systems. The results obtained show that our seed scheduling and task scheduling approaches effectively identify more bugs than the baseline in cloud-native applications. This implies that our proposed solution skilfully balances exploration and exploitation. As a result, CloudFuzz outperforms the baseline in detecting unique bugs in critical cloud-native applications.

Performance overhead in CloudFuzz

Results on real-world applications show that the overhead of our AMAB-based seed selection and task scheduling solution is acceptable. Surprisingly, CloudFuzz is slightly more efficient and better than go-fuzz in terms of overhead and throughput when we enable only the smart seed selection optimization on CloudFuzz. When CloudFuzz is set to enable both seed selection and task scheduling optimizations, CloudFuzz introduces only a means of 15% runtime overhead. The CloudFuzz's overhead is very competitive considering its excellent coverage improvement effects.

Performance overhead is a important factor for fuzzers as it directly determines the fuzzing throughput. In essence, it determines the level of iteration achieved during the fuzzing process. In CloudFuzz, the performance overhead comes from two sources: 1) collecting and computing more accurate and finer-grained coverage measurements which are required by our coverage metric and 2) measuring and computing gains and probabilities for each seed and updating the energies for different tasks. In addition, our seed selection and task scheduling strategies improve the execution efficiency of the CloudFuzz process, which offsets some of the overhead from the above aspects. Therefore, when enabling seed selection and/or task scheduling optimization, we should consider the overall overhead and throughput of CloudFuzz at this point, rather than just the overhead that comes with the optimization.

Overhead of coverage metrics

To quantify the positive and negative impact on performance overhead for our AMAB-based solution, we should understand the overhead difference between the coverage metric of go-fuzz and CloudFuzz. First, we investigated the proportion of the overhead caused by our fine-grained coverage metric. The results on targets of our dataset are shown in Fig. 5a, where the x-axis represents the individual runs of the targets (10x×16=160 in total) and the y-axis represents the overhead proportion of the coverage metrics applied by the fuzzers. For most targets, our coverage metric solution introduces a runtime overhead of less than 40%. For about one in five targets, the overhead proportion for our more accurate and finer-grained coverage metric is above 60%, but below 140%. The average of the overhead proportion for our coverage metric is around 35%, as shown in the box plot on the right-hand side of Fig. 5a.

Overhead of seed selection

Next, we measured the performance overhead of CF-seed versus go-fuzz on Table 2 by the throughput of the fuzzers. Figure 5b shows the ratio of the increase in the overhead of CF-seed in an ascending order, which is the difference in throughput between go-fuzz and CF-seed as a proportion of the throughput of go-fuzz. It is worth noting that if the throughput of CF-seed is greater than the average throughput of go-fuzz, then the difference between the average throughput of go-fuzz minus the throughput of CF-seed will be a negative number, in which case the increased overhead of CF-seed will show up as a negative number based on our calculation.

As shown in Fig. 5b, the throughput of CF-seed exceeds that of go-fuzz on a third of the targets. More specifically, CF-seed has a throughput improvement of more than 50% on about 10% of the targets, and improves runtime performance by more than 25% on a fifth of the targets. The total overhead of CF-seed is less than 25% on more than 80% of the targets. The average of overhead ratio for CF-seed is negative 3%, as shown in the box plot of Fig. 5b. This means that CF-seed achieves better overall throughput in 160 runs of 16 real-world targets.

Overhead of task scheduling

Similar to the overhead measurement for CF-seed, we measured the overhead of CloudFuzz, which enables two AMAB-based optimizations. Figure 5c shows the overhead ratio of CloudFuzz over 160 runs, which is calculated in the same way as Fig. 5b. We can see that CloudFuzz achieved more throughput than go-fuzz on about 30% of the targets. And CloudFuzz introduced below 25% runtime overhead on half of the targets. On the remaining targets, CloudFuzz bring about more overhead than CF-seed. CloudFuzz increased overhead by more than 50% on a third of the targets. The average overhead ratio of CloudFuzz is about 14%, which is higher than the overhead ratio of CF-seed but lower than that of go-fuzz-well. It is easy to see that our task scheduling optimization needed to measure and record finer-grained coverage information at each stage of fuzzing, which caused more overhead than the seed selection optimization. According to the average overhead caused by the seed selection optimization, the average overhead caused by the task scheduling optimization is approximately 17%.

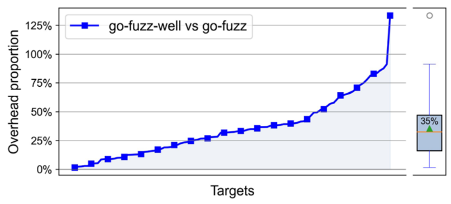
Effectiveness of smart seed selection

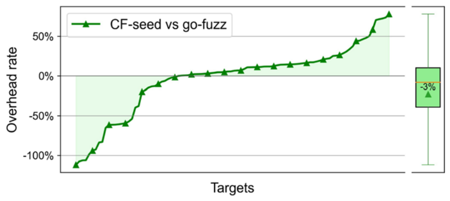
Experiment results on dataset I show that our smart seed selection optimization significantly improves the capability and efficiency of fuzzing. More specifically, our smart seed selection optimization helps the fuzzer to achieve higher code coverage faster and reducees the number of seed candidates caused by highly sensitive coverage metrics.

Code coverage

First, we evaluate the impact of smart seed selection on code coverage. Figure 4 illustrates the coverage growth trends through the green curves marked as CF-seed. Intuitively, CF-seed achieves higher coverage than go-fuzz across all targets. Table 4 Columns 2 and 3 list the maximum coverage achieved by CF-seed and the specific coverage increase rates compared with go-fuzz, respectively. The results demonstrate a consistent increase in maximum coverage, thereby evidencing the enhanced effectiveness of the smart seed selection optimization. With a few exceptions, CF-seed shows significant gains over go-fuzz, as is particularly evident in test cases such as No. 2 and No. 14, where the increase exceeds 90%.

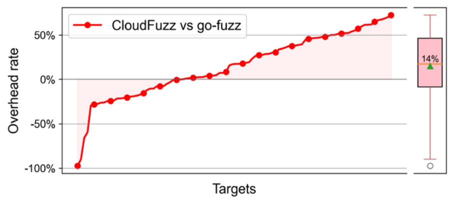
On the other hand, CF-seed achieves slightly lower maximum coverage than CloudFuzz on three out of four of the targets. However, CF-seed achieves higher coverage than CloudFuzz on targets No. 2 and No. 16, possibly due to higher throughput. As previously mentioned, CF-seed also achieves a higher throughput than both go-fuzz and CloudFuzz. For these targets, the coverage gains derived from task scheduling optimization are less than the detrimental effects caused by the task scheduling





(a) Overhead proportion caused by our accurate edge coverage metric by comparing the throughput between go-fuzz and go-fuzz-well.

(b) Comparison between throughput of CF-seed and go-fuzz running 10 times on targets in Table 2.



(c) Comparison between throughput of CloudFuzz and go-fuzz running 10 times on targets in Table 2.

Fig. 5 Performance overhead rate caused by different optimizations of CloudFuzz

overhead. In the case of target No. 13, CF-seed achieved the same coverage as CloudFuzz, but CloudFuzz reached the coverage upper limit faster. In terms of code coverage, our smart seed selection optimization enables the fuzzer to discover more interesting states and explore wider regions of the code, thereby increasing the likelihood of finding more bugs in the targets.

Seed in the corpus

Second, we investigate the number of seeds generated by the fuzzers to deduce whether the smart seed selection optimization can find more execution states than the baseline and can also inhibit the seed explosion problem caused by highly sensitive coverage metrics. Figure 6 shows the statistics of the seeds generated by the fuzzer over 160 (16×10) runs in the targets of Table 2, and we have labelled the average of them. On the one hand, the number of seeds in CF-seed's corpus is almost 3 times than that of go-fuzz, illustrating that CF-seed can trigger more execution states than go-fuzz. Similarly, go-fuzz-well produced almost 4 times as many seeds as go-fuzz. However, given the inferior throughput of go-fuzz-well, too many seeds put a heavy load on a fuzzer's scheduler. On the other hand, CF-seed reduced the number of seed candidates by about 22%, mitigating the burden on the scheduler. From the results of 24 hours, we can see that CF-seed has solved the seed explosion problem that can occur in go-fuzz-well. Through the above analysis of seeds generated by fuzzers, we explain that our smart seed selection optimization can inhibit the seed explosion problem caused by high-sensitive coverage metrics.

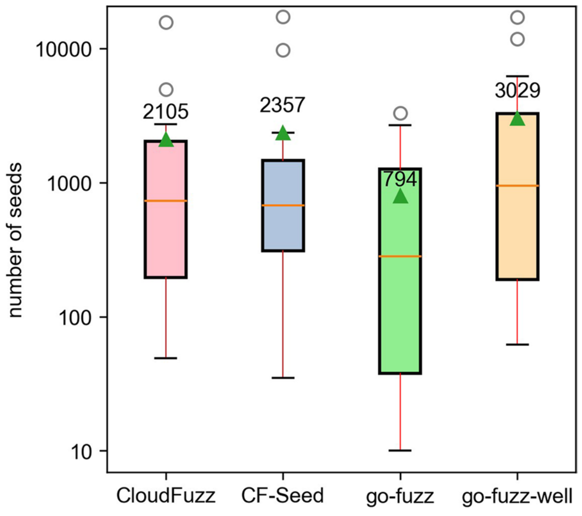


Fig. 6 Average number of seeds generated by different fuzzers on the 16 targets in Table 2

Effectiveness of adaptive task scheduling

Experiments results on our dataset show that our adaptive task scheduling optimization significantly improves the upper bound of code coverage. Furthermore, compared to CF-seed, seeds generated by CloudFuzz optimized with adaptive task scheduling achieve higher code coverage at the same cost.

Code coverage

Table 6 shows the statistics of the target applications tested by CF-seed (denoted as CFs) and CloudFuzz (denoted as CF). First, we compare the coverage increment to evaluate the impact of adaptive task scheduling in terms of code coverage. On most of targets (14/16), the coverage of CloudFuzz is better than that of CF-seed. It means that our adaptive task scheduling optimization can help fuzzers to trigger more code based on CF-seed. On half of the targets, our task scheduling optimization could improve coverage by at least 10% over CF-seed with smart seed selection optimization enabled. Compared with CF-seed, the average coverage increment brought by adaptive task scheduling optimization is 23.5%. In particular, CloudFuzz triggered over 60% more code regions than CF-seed on target No 1, No 8. The results of the code coverage experiments show that our task scheduling optimiation using a finer-grained gain estimation solution could more accurately guide the fuzzing process to balance exploration and exploitation. Compared to CloudFuzz, CF-seed only coarsely balances exploration and exploitation. As a result, CloudFuzz outperforms CF-seed in terms of coverage.

Average-cost

As aforementioned, CloudFuzz achieved higher coverage than baseline fuzzers, and improved the overall efficiency of the fuzzing process. Since the seeds have a direct effect on fuzzers, our AMAB-based optimization affects fuzzers by controlling the process associated with seeds. Therefore, we focused on the number of seeds generated by fuzzers and the quality of the seeds to evaluate the efficiency of our task scheduling optimization.

As shown in Fig. 5c, though task scheduling optimization, CloudFuzz further mitigates the number of seeds to solve the seed explosion problem. In total, compared to go-fuzz-well, CloudFuzz's solution reduced the number of seeds by 41%. Compared to CF-seed, our adaptive task schedluing optimization reduces the number of generated seeds by about 10%. Our task scheduling optimization has a positive impact on controlling the size of the corpus. In order to quantitatively assess the quality of seeds generated by fuzzers, we calculated the average-cost of the CloudFuzz (denoted as AC CF) and that of CF-seed (denoted as AC CFs) in different targets. The

Table 6 Maximum coverage and average-cost value of CF-seed and CloudFuzz on targets in Table 2

No	Cov CFs	Cov CF	Seeds CFs	Seeds CF	AC CFs	AC CF
No 1	2676	4318(61.36%)	35	62	76.45	69.65(-8.9%)
No 2	5950	5658(-4.91%)	2025	2726	2.93	2.08(-29.16%)
No 3	6381	6877(7.77%)	2344	2778	2.72	2.48(-8.99%)
No 4	7551	8421(11.52%)	184	104	41.03	80.97(97.35%)
No 5	991	1073(8.27%)	288	164	3.44	6.54(90.19%)
No 6	1743	1924(10.38%)	233	615	7.48	3.13(-58.18%)
No 7	24559	25932(5.59%)	9696	4933	2.53	5.26(107.54%)
No 8	4869	11509(136.37%)	515	226	9.45	50.92(438.64%)
No 9	4741	6957(46.74%)	952	1733	4.98	4.01(-19.39%)
No 10	6316	6643(5.18%)	1237	1875	5.11	3.54(-30.61%)
No 11	3118	4135(32.62%)	1764	1278	1.77	3.24(83.05%)
No 12	52694	66631(26.45%)	17180	15720	3.07	4.24(38.19%)
No 13	69555	70762(1.74%)	838	738	83.00	95.88(15.52%)
No 14	3468	3671(5.85%)	404	49	8.58	74.92(772.75%)
No 15	5683	7249(27.56%)	55	53	103.33	136.77(32.37%)
No 16	4289	4003(-6.67%)	335	278	12.80	14.40(12.47%)

The 2nd and 3rd columns of this table show the maximum coverage achieved by CF-Seed and CloudFuzz respectively. The 4th and 5th columns show the number of new seeds discovered by CF-Seed and CloudFuzz respectively, while the last two columns illustrate the average costs for CF-Seed and CloudFuzz

average-cost is defined as the average coverage improved by a single seed. In the last column of Table 6, it shows the average-cost of CloudFuzz and demonstrating the improvement of average-cost compared to CF-seed. For more than 60% of the targets, the average-cost of CloudFuzz is better than that of CF-seed. On target No 7, No 8, and No 14, the average-cost of CloudFuzz was more than 100% and even 770% than that of CF-seed. Although CF-seed outperformed CloudFuzz on average cost for some targets, CloudFuzz achieved better coverage.

Related work

Coverage-guided greybox fuzzing

Coverage-guided fuzzing is one of the most effective and popular techniques for finding bugs in practice. AFL [1], LIBFUZZER [25] and honggfuzz [26] are widely used in the OSS-Fuzz [27] service, which has detected over 8800 vulnerabilities and 28800 bugs across 850 open source projects. As its effectiveness and simplicity, coverage-guided fuzzing has been optimized by academic research in several areas. On the one hand, some work has applied program analysis to understand the behaviour of tested targets, which helps fuzzing to adapt different programs. For instance, taint analysis [16, 20, 28], concolic execution [6, 29, 30], static analysis [31–33], deep learning [34, 35] and reinforcement learning [11, 22, 36] are used to boost fuzzer performance. On the other hand, some work has attempted to transform fuzzing to better test specific types of targets, such as JIT compilers [37–40], OS kernel [23, 41–43], protocol [44, 45], rounter [46, 47], and smart contracts [48, 49]. For example, to find JIT compiler vulnerabilities, some fuzzers use an abstract syntax tree to represent and generate JavaScript code as seeds. CloudFuzz utilizes an accurate fine-grained coverage metric, which has been proven to be more effective in improving fuzzer's performance. Meanwhile, for the cloud-native application of Golang targets, CloudFuzz proposes an AMAB-based seed and task scheduler to help fuzzing balance exploration and exploitation in large targets.

Coverage metric for fuzzing

Coverage metrics as one of the most important factors for coverage-guided fuzzing, which guides fuzzers to generate and select proper seeds to test as much of the code as possible. VUzzer [16] considered fundamental properties of the application as a part of the coverage metric to implement an application-aware strategy to maximize coverage. However, the block coverage metric used by VUzzer provides much less information than edge coverage metrics and cannot even distinguish between blocks with different predecessors. AFL [1] achieved an efficient edge coverage metric, making AFL as the most successful fuzzer. Thus, a lot of work have improved coverage metrics based on the edge coverage metric. Angora [20] found the edge coverage metric of AFL is context-insensitive, and proposed a context-sensitive edge coverage metric to distinguish the same edge in different contexts. Matryoshka [31] combined the edge coverage metric with different types of conditional statements to find inputs that satisfy deeply nested branches. However, they did not implement a solution to address the potential seed explosion problem when applying the more sensitive coverage metrics. CollAFL [13] found the hash collision problem in AFL and proposed an accurate solution for edge coverage metrics, and their results showed a significant improvement. FAIRFUZZ [50] defined the concept of rare branches that are rarely hit by inputs, and guided fuzzing through the rare branch coverage metric. Jiang, et al. [21] proposed a context-sensitive concurrency coverage metric to detect specific data races that are difficult to find using context-insensitive coverage. Wang, et al. [15] proposed a coverage accounting based on memory operations to evaluate security impacts of coverage. These papers introduced different factors of bug detection and code exploration and demonstrated these coverage metrics are effective.

Wang, et al. [10] evaluated multiple coverage metrics and showed that there is no grand slam coverage metric that can beat others. The normal edge coverage metric can still beat more sensitive coverage metrics on some targets. Thus, considering the overhead of additional analysis by more sensitive edge coverage metrics, CloudFuzz employs an accurate edge coverage metric for large cloud-native applications. Our coverage metrics strike a balance between the coverage information and coverage measurement overhead. Wang, et al. [11] proposed a multi-level coverage metric to collect coverage information and introduced the MAB model to mitigate the seed explosion problem. It was shown that fuzzing should consider scheduling algorithms, especially when highly sensitive coverage metrics are applied. This is because more sensitive coverage metrics introduced more seeds and overhead, which created a more serious seed explosion problem. Therefore, CloudFuzz introduces the AMAB-based scheduling algorithm to mitigate too many seeds needed to handle.

Improving power scheduling

Some work has paid attention to power scheduling, assigning different powers to seeds and tasks, which helps fuzzing make the most correct decision at different phases. AFLFAST [18] regarded the fuzzing process as a Markov chain model and assigned more power to seeds that triggered low-frequency paths. Similarly, VUzzer [16] observed that a large percentage of seeds fell into the error-handling code and allocated less power to these seeds, reducing the energy of high-frequency paths. Mopt [51] focused on the scheduling strategies for mutation operations, selecting different mutation operations for different seeds. CollAFL [13] prioritized the seeds with untouched neighbour descendants to mutate, which has a higher probability of exploring untouched paths.

Woo, et al. [52] modelled the scheduling problem as an instance of the classical Multi-Armed Bandit (MAB) problem to maximize the number of unique bugs in time. However, this work ignored the fact that in the classic MAB the number of arms is fixed, but the number of seeds is increased during fuzzing, which is different from the classic MAB problem. Wang, et al. [11] also modelled the seed scheduling problem as the classical MAB problem, but they introduced the rareness of seeds as a factor in the reward calculation to resolve the above contradiction. EcoFuzz [22] used a variant of the Adversarial MAB model to schedule seeds, but prioritized exploitation rather than balancing exploration and exploitation, thus failing to address the seed exploration problem. Considering the features of kernel fuzzing, Syzvegas [23] applied the AMAB model for seed scheduling and task scheduling stages respectively. MobFuzz [36] modeled multiple objective optimizations as a multi-player multi-armed bandit problem to allocate energy for objective combination and seeds.

Conclusion

Coverage metrics play an important role in coverage-guided fuzzing. Accurate and fine-grained coverage metrics could help fuzzing to find more bugs and achieve higher coverage. Unfortunately, fine-grained coverage metrics lead to more seeds to select, and even seed explosion. Existing fuzzers for Golang still use simple and coarse-grained block coverage metrics, which hinders detect bugs in cloud-native applications written with Golang. In this work, we apply an accurate edge coverage metric with go-fuzz to achieve fine-grained testing for cloud-native applications. To mitigate the seed explosion problem caused by fine-grained coverage metrics and large targets, we propose smart seed selection and adaptive task scheduling algorithm based on a variant of the classical AMAB algorithm. The results of evaluation experiments show that our approach significantly outperforms go-fuzz on real-world targets.

Abbreviations

AMAB	Adversarial multi-armed bandit
MAB	Multi-armed bandit

Acknowledgements
We sincerely thank the Reviewers and the Editor for their valuable suggestions.

Authors' contributions

Jiageng Yang wrote the main manuscript text and Chuanyi Liu proposed the method and revised the manuscript. All authors reviewed the manuscript.

Availability of data and materials

All fuzzer drivers and the dataset used in this paper are available from: https://github.com/yangjiageng/CloudFuzz_FuzzDriver.

Declarations

Ethics approval and consent to participate
This article does not contain any studies with human participants or animals performed by any of the authors.

Consent for publication
The authors read and approved the final manuscript.

Competing interests
The authors declare no competing interests.

Received: 19 February 2024 Accepted: 17 June 2024
Published online: 26 June 2024

References

1. Zalewski M (2014) American fuzzy lop. https://lcamtuf.coredump.cx/afl. Accessed 17 Feb 2024
2. Vyukov D (2021) go-fuzz: randomized testing for go. https://github.com/dvyukov/go-fuzz. Accessed 17 Feb 2024
3. Wilk J (2019) Python-afl. https://jwilk.net/software/python-afl. Accessed 17 Feb 2024
4. Jiang J, Xu H, Zhou Y (2021) Rulf: Rust library fuzzing via api dependency graph traversal. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, Melbourne, pp 581–592
5. Google (2021) Go fuzzing. https://go.dev/security/fuzz. Accessed 17 Feb 2024
6. Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T (2019) Redqueen: Fuzzing with input-to-state correspondence. In: Network and Distributed Systems Security (NDSS) Symposium, vol 19. The Internet Society, San Diego, pp 1–15
7. Fioraldi A, Maier D, Eißfeldt H, Heuse M (2020) Afl++ combining incremental steps of fuzzing research. In: Proceedings of the 14th USENIX Conference on Offensive Technologies. {USENIX} Association, pp 10–10
8. Pham VT, Böhme M, Santosa AE, Căciulescu AR, Roychoudhury A (2019) Smart greybox fuzzing. IEEE Trans Softw Eng 47(9):1980–1997
9. Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T (2017) kafl: Hardware-assisted feedback fuzzing for os kernels. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, pp 167–182
10. Wang J, Duan Y, Song W, Yin H, Song C (2019) Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In: Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19). USENIX Association, Beijing, pp 1–15
11. Wang J, Song C, Yin H (2021) Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In: Network and Distributed Systems Security (NDSS) Symposium, The Internet Society, virtual, vol 2021
12. Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z (2020) Path sensitive fuzzing for native applications. IEEE Trans Dependable Secure Comput 19(3):1544–1561
13. Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z (2018) Collafl: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP). {IEEE} Computer Society, San Francisco, pp 679–696
14. Auer P, Cesa-Bianchi N, Freund Y, Schapire RE (1995) Gambling in a rigged casino: The adversarial multi-armed bandit problem. In: Proceedings of IEEE 36th annual foundations of computer science. {IEEE} Computer Society, Milwaukee, pp 322–331
15. Wang Y, Jia X, Liu Y, Zeng K, Bao T, Wu D, Su P (2020) Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In: Network and Distributed Systems Security (NDSS) Symposium. The Internet Society, San Diego
16. Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H (2017) Vuzzer: Application-aware evolutionary fuzzing. In: Network and Distributed Systems Security (NDSS) Symposium. The Internet Society, San Diego, vol 17. pp 1–14
17. Böhme M, Pham VT, Nguyen MD, Roychoudhury A (2017) Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, pp 2329–2344
18. Böhme M, Pham VT, Roychoudhury A (2016) Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, pp 1032–1043
19. Coppik N, Schwahn O, Suri N (2019) Memfuzz: Using memory accesses to guide fuzzing. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). IEEE, Xi'an, pp 48–58
20. Chen P, Chen H (2018) Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP). {IEEE} Computer Society, San Francisco, pp 711–725
21. Jiang ZM, Bai JJ, Lu K, Hu SM (2022) Context-sensitive and directional concurrency fuzzing for data-race detection. In: Proceedings of the 29th Network and Distributed System Security Symposium (NDSS). The Internet Society, San Diego, pp 1–18
22. Yue T, Wang P, Tang Y, Wang E, Yu B, Lu K, Zhou X (2020) Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: Proceedings of the 29th USENIX Conference on Security Symposium. USENIX Association, Boston, pp 2307–2324
23. Wang D, Zhang Z, Zhang H, Qian Z, Krishnamurthy SV, Abu-Ghazaleh NB (2021) Syzvegas: Beating kernel fuzzing odds with reinforcement learning. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, Vancouver, pp 2741–2758
24. Jiang, Zu-Ming and Bai, Jia-Ju and Lu, Kangjie and Hu, Shi-Min (2020) Fuzzing error handling code using context-sensitive software fault injection. In: Proceedings of the 29th USENIX Conference on Security Symposium. USENIX Association, Boston, pp 2595–2612
25. Google (2015) Libfuzzer: a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed 17 Feb 2024
26. Swiecki R (2017) Honggfuzz. https://honggfuzz.dev. Accessed 17 Feb 2024
27. Google (2016) Oss-fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz. Accessed 17 Feb 2024
28. Wang T, Wei T, Gu G, Zou W (2010) Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy. {IEEE} Computer Society, Oakland, pp 497–512
29. Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G (2016) Driller: Augmenting fuzzing through selective symbolic execution. In: Network and Distributed Systems Security (NDSS) Symposium, vol 16. The Internet Society, San Diego, pp 1–16
30. Zhao L, Duan Y, Yin H, Xuan J (2019) Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In: Network and Distributed Systems Security (NDSS) Symposium. The Internet Society, San Diego, pp 1–18
31. Chen P, Liu J, Chen H (2019) Matryoshka: fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, pp 499–513
32. Han H, Cha SK (2017) Imf: Inferred model-based fuzzer. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, pp 2345–2358
33. Peng H, Shoshitaishvili Y, Payer M (2018) T-fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP). {IEEE} Computer Society, San Francisco, pp 697–710
34. Godefroid P, Peleg H, Singh R (2017) Learn &fuzz: Machine learning for input fuzzing. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). {IEEE} Computer Society, Urbana, pp 50–59
35. She D, Pei K, Epstein D, Yang J, Ray B, Jana S (2019) Neuzz: Efficient fuzzing with neural program smoothing. In: 2019 IEEE Symposium on Security and Privacy (SP). {IEEE} Computer Society, San Francisco, pp 803–817
36. Zhang G, Wang P, Yue T, Kong X, Huang S, Zhou X, Lu K (2022) Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In: Network and Distributed Systems Security (NDSS) Symposium, vol 16. The Internet Society, San Diego, pp 1–18
37. Bernhard L, Scharnowski T, Schloegel M, Blazytko T, Holz T (2022) Jit-picking: Differential fuzzing of javascript engines. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22. Association for Computing Machinery, New York, pp 351–364

38. Groß S, Koch S, Bernhard L, Holz T, Johns M (2023) Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In: Network and Distributed Systems Security (NDSS) Symposium. vol 2023. The Internet Society, San Diego, pp 1–18
39. Wang J, Chen B, Wei L, Liu Y (2017) Skyfire: Data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP). {IEEE} Computer Society, San Jose, pp 579–594
40. Wang J, Zhang Z, Liu S, Du X, Chen J (2023) Fuzzjit: Oracle-enhanced fuzzing for javascript engine jit compiler. In: Proceedings of the 32nd USENIX Conference on Security Symposium. USENIX Association, Anaheim, CA, pp 1865–1882
41. Corina J, Machiry A, Salls C, Shoshitaishvili Y, Hao S, Kruegel C, Vigna G (2017) Difuze: Interface aware fuzzing for kernel drivers. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, pp 2123–2138
42. Google (2015) syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller. Accessed 17 Feb 2024
43. Xu W, Moon H, Kashyap S, Tseng PN, Kim T (2019) Fuzzing file systems via two-dimensional input space exploration. In: 2019 IEEE Symposium on Security and Privacy (SP). {IEEE} Computer Society, San Francisco, pp 818–834
44. Banks G, Cova M, Felmetsger V, Almeroth K, Kemmerer R, Vigna G (2006) Snooze: toward a stateful network protocol fuzzer. In: Information Security, vol 4176. Springer Berlin Heidelberg, Berlin, pp 343–358
45. Maier D, Bittner O, Munier M, Beier J (2022) Fitm: Binary-only coverage-guided fuzzing for stateful network protocols. In: Workshop on Binary Analysis Research (BAR), vol 2022. The Internet Society, San Diego, pp 1–11
46. Chen L, Wang Y, Cai Q, Zhan Y, Hu H, Linghu J, Hou Q, Zhang C, Duan H, Xue Z (2021) Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In: USENIX Security Symposium. {USENIX} Association, pp 303–319
47. Qin C, Peng J, Liu P, Zheng Y, Cheng K, Zhang W, Sun L (2023) Ucrf: Static analyzing firmware to generate under-constrained seed for fuzzing soho router. Computers & Security, vol 128, pp 103–157
48. Ma F, Chen Y, Ren M, Zhou Y, Jiang Y, Chen T, Li H, Sun J (2023) Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols. In: Network and Distributed Systems Security (NDSS) Symposium, vol 2023. The Internet Society, San Diego, pp 1–18
49. Zuo F, Luo Z, Yu J, Liu Z, Jiang Y (2021) Pavfuzz: State-sensitive fuzz testing of protocols in autonomous vehicles. In: 2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, San Francisco, pp 823–828
50. Lemieux C, Sen K (2018) Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM, Montpellier, pp 475–485
51. Lyu C, Ji S, Zhang C, Li Y, Lee WH, Song Y, Beyah R (2019) Mopt: Optimized mutation scheduling for fuzzers. In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, pp 1949–1966
52. Woo M, Cha SK, Gottlieb S, Brumley D (2013) Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. Association for Computing Machinery, New York, pp 511–522