Abstract—Coverage metrics play an essential role in greybox fuzzing. Recent work has shown that fine-grained coverage metrics could allow a fuzzer to detect bugs that cannot be covered by traditional edge coverage. However, fine-grained coverage metrics will also select more seeds, which cannot be efficiently scheduled by existing algorithms. This work addresses this problem by introducing a new concept of multi-level coverage metric and the corresponding reinforcement-learning-based hierarchical scheduler. Evaluation of our prototype on DARPA CGC showed that our approach outperforms AFL and AFLFast significantly: it can detect 20% more bugs, achieve higher coverage on 83 out of 180 challenges, and achieve the same coverage on 60 challenges. More importantly, it can detect the same number of bugs and achieve the same coverage faster. On FuzzBench, our approach achieves higher coverage than AFL++ (Qemu) on 10 out of 20 projects.

I. INTRODUCTION

Greybox fuzzing is a state-of-the-art testing technique that has been widely adopted by the industry and has successfully found tens of thousands of vulnerabilities in widely used software. For example, the OSS-Fuzz project has found more than 10,000 bugs in popular open-sourced projects like OpenSSL since its launch in December 2016.

As shown in Figure 1, greybox fuzzing can be modeled as a genetic process where new inputs are generated through mutation and crossover/splice. The generated inputs are selected according to a fitness function. Selected inputs are then added back to the seed pool for future mutation. Unlike natural evolution, due to the limited processing capability, only a few inputs from the seed pool will be scheduled to generate the next batch of inputs. For example, a single fuzzer instance can only schedule one seed at a time.

The most common fitness function used by off-the-shelf fuzzers like American Fuzzy Lop (AFL) [55] is edge coverage, i.e., inputs that cover new branch(es) will be added to the seed pool, as its goal is to achieve higher edge coverage of the code. While most fuzzers are coverage-guided (i.e., use new coverage as the fitness function), recent research has shown that the genetic process can also be used to discover a diversity of program properties by using a variety of fitness functions [26], [32], [34].

An important property of a fitness function (e.g., a coverage metric) is its ability to preserve intermediate waypoints [32]. To better illustrate this, consider flipping a magic number check $a = 0x\text{deadbeef}$ as an example. If a fuzzer only considers edge coverage, then the probability of generating the correct $a$ with random mutations is $2^{32}$. However, if the fuzzer can preserve important waypoints, e.g., by breaking the 32-bit magic number into four 8-bit number [25], then solving this checking will be much more efficient since the answer can be generated from a sequence as $0xef$, $0xbeef$, and $0xadbeef$. This check can also be solved faster by understanding distances between current value of $a$ and the target value $0xef$, $0xbeef$, and $0xadbeef$. More importantly, recent research has shown that many program states cannot be reached without saving critical waypoints [30], [45].

Wang et al. [45] formalize the ability to preserve intermediate waypoints as the sensitivity of a coverage metric. Conceptually, a more sensitive metric would lead to more program states (e.g., code coverage). However, the empirical evaluation of [45] shows that this is not always the case. The reason is that, a more sensitive coverage metric will select more seeds, which could cause seed explosion and exceed the fuzzer’s ability to schedule. As a result, many seeds may never be scheduled or be scheduled without enough time/power to make a breakthrough [9].

In this work, we aim to address the seed explosion problem with a hierarchical scheduler. Specifically, fuzzing can be modeled as a multi-armed bandit (MAB) problem [49], where the scheduler needs to balance between exploration and exploitation. With a more sensitive coverage metric like branch distance, exploitation can be considered as focusing on solving a hard branch (e.g., magic number check), and exploration can
be considered as exercising an entirely different function. Our crucial observation is that when a coverage metric $C_j$ is more sensitive than $C_i$, we can use $C_j$ to save all the intermediate waypoints without losing the ability to discover more program states; but at the same time, we can use $C_i$ to cluster seeds into a representative node and schedule at node level to achieve better exploration. More specifically, the scheduler will choose a node first, and then choose a seed in that node. Based on this observation, we propose to organize the seed pool as a multi-level tree where leaf nodes are real seeds and internal nodes are less sensitive coverage measurements. The closer a node is to the leaf, the more sensitive the corresponding coverage measurement is. Then we can utilize the existing MAB algorithms to further balance between exploitation and exploration.

To validate our idea, we implemented two prototypes; one AFL-HIER based on AFL and the other AFL++-HIER based on AFL++. We performed extensive evaluation on the DARPA Cyber Grand Challenge (CGC) dataset [10] and Google FuzzBench [21] benchmarks. Compared to AFLFAST [9], AFL-HIER can find more bugs in CGC (77 vs. 61). AFL-HIER also achieved better coverage in about 83 of 180 challenges and the same coverage on 60 challenges. More importantly, AFL-HIER can find the same amount of bugs and achieve the same coverage faster than AFLFAST. On FuzzBench, AFL++-HIER achieved higher coverage on 10 out of 20 projects than AFL++ (Qemu).

Contributions. This paper makes the following contributions:

- We propose multi-level coverage metrics that bring a novel approach to incorporate sensitive coverage metrics in greybox fuzzing.
- We design a hierarchical seed scheduling algorithm to support the multi-level coverage metric based on the multi-armed bandits model.
- We implement our approach as an extension to AFL and AFL++ and release the source code at [https://github.com/bitsecurerlab/aflplusplus-hier](https://github.com/bitsecurerlab/aflplusplus-hier)
- We evaluate our prototypes on DARPA CGC and Google FuzzBench. The results show that our approach not only can trigger more bugs and achieve higher code coverage, but also can achieve the same coverage faster than existing approaches.

## II. BACKGROUND

### A. Greybox Fuzzing

Algorithm 1 illustrates the greybox fuzzing process in more detail. Given a program to fuzz and a set of initial seeds, the fuzzing process consists of a sequence of loops named rounds. Each round starts with selecting the next seed for fuzzing from the pool according to the scheduling criteria. The scheduled seed is assigned to a certain amount of power that determines how many new test cases will be generated in this round. Next, test cases are generated through (random) mutation and crossover based on the scheduled seed. Compared to blackbox and whitebox fuzzing, the most distinctive step of greybox fuzzing is that, when executing a newly generated input, the fuzzer uses lightweight instrumentations to capture runtime features and expose them to the fitness function to measure the "quality" of a generated test case. Test cases with good quality will then be saved as a new seed into the seed pool. This step allows a greybox to gradually evolve towards a target (e.g., more coverage). The effectiveness and efficiency of greybox fuzzing depend on the following factors.

![Algorithm 1: Greybox Fuzzing Algorithm](image)

Test case measurement. As a genetic process, the fitness function of the fuzzer decides what kind of program properties the fuzzer can discover [32]. While fuzzing has been successfully applied to many different domains in recent years with different fitness functions, the most popular one is still code coverage (i.e., a test case that triggers new coverage will be saved as a new seed). However, coverage measurements can be diverse. Notably, AFL++ [55] measures the edge coverage of test cases. More precisely, it maintains a global map where the hashed value of an edge (i.e., the pair of the current basic block address and the next basic block address) is used as an indexing key to access the hit_count of the edge, which records how many times it has been taken so far. The hit counts are bucketed into small powers of two. After a test case completes its execution, the global hit_count map will be updated according to its edges, and it will be selected as a new seed if a new edge is found or the hit count of one edge increases into a new bucket. As we can see, this measurement does not consider the order of edges and can miss interesting seeds is the hash of a new edge collides with the hash of an already covered edge [19].

Seed scheduling criteria. The limited processing capability makes it essential to prioritize some seeds over others in order to maximize the coverage. For example, AFL++ prefers seeds with small sizes and short execution time to achieve a higher fuzzing throughput. Furthermore, it maintains
high-frequency paths tend to be covered by invalid test cases. 

Fuzzing throughput. Fuzzing throughput is another critical factor that decides how fast a fuzzer can discover new coverage. AFL [55] uses the fork server and persistent mode to reduce initialization overhead, thus improving the throughput. Xu et al. [51] proposed new OS primitives to improve fuzzing throughput further. FirmaFL [57] uses augmented emulation to speed-up fuzzing firmware. Because high throughput is the key factor that allows greybox fuzzers to beat whitebox fuzzers in practice, one must pay special attention to the trade-off between throughput and the above three factors (coverage measurement, scheduling algorithm, and mutation strategy). That is, improvements of the above three factors at the cost of throughput may unexpectedly result in worse fuzzing performance.

B. Multi-Armed Bandit Model

The multi-armed bandit model offers a fundamental framework for algorithms that learn optimal resource allocation policies over time under uncertainty. The term “bandit” comes from a gambling scenario where the player faces a row of slot machines (also known as one-armed bandits) yielding random payoffs and seeks the best strategy of playing these machines to gain the highest long-term payoffs.

In the basic formulation, a multi-player bandit problem is defined as a tuple $(A, R)$, where $A$ is a known set of $K$ arms (or actions) and $R = (r[a])$ is an unknown but fixed probability distribution over rewards. At each time step $t$ the agent selects an arm $a_t$, and observes a reward $r_t = R[a_t]$. The objective is to maximize the cumulative rewards $\sum_{t=1}^{T} r_t$.

Initially, the agent has no information about which arm is expected to have the highest reward, so it tries some randomly and observes the rewards. Then the agent has more information than before. However, it has to face the trade-off between “exploitation” of the arm that is with the highest expected reward so far, and “exploration” to obtain more information about the expected rewards of the other arms so that it does not miss out on a valuable one by simply not trying it enough times.

Various algorithms are proposed to make the optimal trade-off between exploitation and exploration of arms. Upper Confidence Bound (UCB) algorithms [5] are a family of bandit algorithms that perform impressively. Specifically, they construct a confidence interval to estimate each arm’s true reward, and select the arm with the highest UCB each time. Notably, the confidence interval is designed to shrink when the arm with its reward is sampled more. As a result, while the algorithm tends to select arms with high average rewards, it will periodically try less explored arms since their estimated rewards have wider confidence intervals.

Take UCB1 [2], which is almost the most fundamental one, as an example. It starts with selecting each arm once to obtain an initial reward. Then at each time step, it selects arm $a$ that maximizes $Q(a) + C \times \frac{\log(N)}{n_a}$ where $Q(a)$ is the average reward obtained from arm $a$, $C$ is a predefined constant that is usually set to $\sqrt{2}$, $N$ is the overall number of selections done so far, and $n_a$ is the number of times arm $a$ has been selected.

Seed scheduling can be modeled as a multi-armed bandit problem where seeds are regarded as arms [49], [54]. However, to make the fuzzer benefit from this model, such as maximizing the code coverage, we need to design the reward of scheduling a seed carefully.

III. Multi-Level Coverage Metrics

In this section, we discuss what are multi-level coverage metrics and why they are useful for greybox fuzzing.

A. Sensitivity of Coverage Metrics

Given a mutation-based greybox fuzzer, a fuzzing campaign starts with a set of initial seeds. As the fuzzing goes on, more seeds are added into the seed pool through mutating the existing seeds. By tracking the evolution of the seed pool, we can see how each seed can be traced back to an initial seed via a mutation chain, in which each seed is generated from mutating its immediate predecessor. If we consider a bug triggering test case as the end of a chain and the corresponding initial seed as the start, those internal seeds between them serve as waypoints that allow the fuzzer to gradually reduce the search space to find the bug [32].

The coverage metric used by a fuzzer plays a vital role in creating such chains, from two main aspects. First, if the chain terminates earlier before reaching the bug triggering test case, then the bug may never be discovered by the fuzzer. Wang et al. [45] formally model this ability to preserve critical waypoints in seed chains as the sensitivity of a coverage metric. For example, consider the maze game in Listing 1, which is widely used to demonstrate the capability of symbolic execution of exploring program states. In this game, a player needs to navigate the maze via the pair of $(x, y)$ that determines a location for each step. In order to win the game, a fuzzer has to try as many sequences of $(x, y)$ pairs as possible to find the right route from the starting location to the crashing location. This simple program is very challenging for fuzzers using edge coverage as the fitness function, because there are only four branches related to every pair of $(x, y)$, each checking against a relatively simple condition that can be satisfied quite easily. For instance, five different inputs: “a,” “u,” “d,” “l,” and “e” are enough to cover all branches/cases of the switch statement. After this, even if the fuzzer can generate new interesting
inputs that indeed advance the program’s state towards the goal (e.g., “dd”), these inputs will not be selected as new seeds because they do not provide new edge coverage. As a result, it is extremely hard, if not impossible, for fuzzers that use the edge coverage to win the game [3].

On the contrary, as we will show in §V-G, if a fuzzer can measure the different combinations of $x$ and $y$ (e.g., by tracking different memory accesses via $*(\text{maze } + y + x)$ at line 10), then reaching the winning point will be much easier [3]. Similarly, researchers have also observed that the orderless of branch coverage and hash collisions can cause a fuzzer to drop critical waypoints hence prevent certain code/bugs from being discovered [19], [27], [30].

The second impact of a coverage metric has on creating seed chains is the stride between a pair of seeds in a chain. Specifically, the sensitivity of a coverage metric also determines how likely (i.e., the probability) a newly generated test case will be saved as a new seed. For instance, it is easier for a fuzzer that uses edge coverage to discover a new seed than a fuzzer that uses block coverage. Similarly, as we have discussed in §I, it is much easier to find a match for an 8-bit integer than a 32-bit integer. Böhme et al. [9] model the minimum effort to discover a neighbouring seed as the required power (i.e., mutations). Based on this modeling, a more sensitive coverage metric requires less power to make progress, i.e., a shorter stride between two seeds. Although each seed only carries a small step of progress, the accumulation of them can narrow the search space faster.

While the above discussion seems to suggest that a more sensitive coverage metric would allow fuzzers to detect more bugs, the empirical results from [45] showed this is not always the case. For instance, while memory access coverage would allow a fuzzer to win the maze game (Listing 1), it did not perform very well on many of the DARPA CGC challenges. The reason is that, a more sensitive coverage metric will also create a larger seed pool. As a result, the seed scheduler needs to examine more candidates each time when choosing the next seed to fuzz. In addition to the increased workload of the scheduler, a larger seed pool also increases the difficulty of seed exploration, i.e., trying as many fresh seeds as possible. Since the time of a fuzzing campaign is fixed, more abundant seeds also imply that the average fuzzing time of each seed could be decreased, which could negatively affect seed exploitation, i.e., not fuzzing interesting seeds enough time to find critical waypoints.

Overall, a more sensitive coverage metric boosts the capability (i.e., upper bound) of a fuzzer to explore deeper program states. Nevertheless, in order to effectively utilize its power and mitigate the side effects of the resulting excessive seeds, the coverage metric and the corresponding seed scheduler should be carefully crafted to strike a balance between exploration and exploitation.

B. Seed Clustering via Multi-Level Coverage Metrics

The similarity and diversity of seeds, which can be measured in terms of the exercised coverage, drive the seed exploration and exploitation in a fuzzing campaign. In general, a set of similar seeds gains less information about the program under test than a set of diverse seeds. When a coverage metric measures more fine-grained coverage information (e.g., edge), it can dim the coarse-grained diversity (e.g., block) among different seeds. First, it encourages smaller variances between seeds. Second, it loses the awareness of the potential larger variance among seeds that can be detected by a more coarse-grained metric. For instance, a metric measuring edge coverage is unaware of whether two seeds exercise two different sets of basic blocks or the same set of basic blocks but through different edges. Therefore, it is necessary to illuminate seed similarity and diversity when using a more sensitive coverage metric.

Clustering is a technique commonly used in data analysis to group a set of similar objects. Objects in the same cluster are more similar to each other than to those in a different cluster. Inspired by this technique, we propose to perform seed clustering so that seeds in the same cluster are similar while seeds in different clusters are more diverse. In other words, these clusters offer another perspective that allows a scheduler to zoom in the similarity and diversity among seeds.

Based on the observation that the sensitivity of most coverage metrics for greybox fuzzing can be directly compared (i.e., the more sensitive coverage metric can subsume the less sensitive one), we propose an intuitive way to cluster seeds—using a coarse-grained coverage measurement to cluster seeds selected by a fine-grained metric. That is, seeds in the same cluster will have the same coarse-grained coverage measurement. Moreover, we can use more than one level of clustering to provide more abstraction at the top level and more fidelity at the bottom level. To this end, the coverage metric should allow the co-existence of multiple coverage measurements. We name such a coverage metric a multi-level coverage metric.

Listing 1: A Simple Maze Game
Definition III.2. A coverage metric by entries of the maps.

Definition III.1. A coverage space $\Gamma$ defines the set of enumerable features we pay attention to that can be covered by executing a program.

Some typical coverage spaces are:
- $\Gamma_F$ is the set of all program functions.
- $\Gamma_B$ is the set of all program blocks.
- $\Gamma_E$ is the set of all program edges. Note that an edge is a transition from one block to the next.

It is worth mentioning that in real-world fuzzers such as AFL, the coverage information is recorded via well-crafted hit_count maps. Consequently, the features are signified by entries of the maps.

Definition III.2. A coverage metric $C : (\mathcal{P} \times \mathcal{I}) \rightarrow \Gamma^*$ measures the execution of a program $P \in \mathcal{P}$ with an input $I \in \mathcal{I}$, and produces a set of features that are exercised by it at least once, denoted as $M \in \Gamma^*$.

Since coverage metric is mainly characterized by the coverage space $\Gamma$, it can be simplified with the coverage space. Some typical coverage metrics are:
- $C_F$ measures the functions that are exercised by an execution.
- $C_B$ measures the blocks that are exercised by an execution.
- $C_E$ measures the edges that are exercised by an execution.

Finally, we give the definition of a multi-level coverage metric.

Definition III.3. A coverage metric $C^n : (\mathcal{P} \times \mathcal{I}) \rightarrow (\Gamma^n_1, \ldots, \Gamma^n_n)$ consists of a sequence of coverage metrics $(\Gamma_1, \ldots, \Gamma_n)$. It measures the execution of a program $P \in \mathcal{P}$ with an input $I \in \mathcal{I}$, and produces a sequence of measurements $(M_1, \ldots, M_n)$.

A multi-level coverage metric combines multiple metrics at different levels to assess a seed. As a result, it relies on lower-level coverage measurements to preserve minor variances among seeds so that there will be more abundant seeds in a chain. This helps to reduce the search space of finding bug triggering test cases. Meanwhile, it allows a scheduler to use upper-level measurements to detect major differences among seeds. Note that when $n = 1$, it is reduced to a traditional single level coverage metric.

D. Principles and Examples of Multi-level Coverage Metrics

To further illustrate how a multi-level coverage metric works, we propose some representative examples. We first discuss some principles for developing an effective multi-level coverage metric $C^n \sim (\Gamma_1, \ldots, \Gamma_n)$ for fuzzing a program $P$.

1) Principles: Through the incremental seed clustering, all seeds are put into a hierarchical tree that lays the foundation of our hierarchical seed scheduling algorithm, which will be described in [SIV]. However, the scheduling makes sense only when a node at an upper level can have multiple child nodes at lower levels. This indicates that the cases where if a set of seeds are assessed to be with the same coverage measurement $M_i$, all following measures $M_{i+1}, \ldots, M_n$ will also be the same should be excluded. Motivated by this fundamental requirement, the main principle is that measurements generated by a less sensitive metric should always cluster seeds prior to more sensitive ones. Here, we use the same definition of sensitivity between two coverage metrics as in [43].

Definition III.4. Given two coverage metrics $C_i$ and $C_j$, we say $C_i$ is “more sensitive” than $C_j$, denoted as $C_i \succ C_j$, if

(i) $\forall P \in \mathcal{P}, \forall I_1, I_2 \in \mathcal{I}, C_i(P, I_1) = C_i(P, I_2) \rightarrow C_j(P, I_1) = C_j(P, I_2)$, and

(ii) $\exists P \in \mathcal{P}, \exists I_1, I_2 \in \mathcal{I}, C_j(P, I_1) = C_j(P, I_2) \land C_i(P, I_1) \neq C_i(P, I_2)$

Specifically, take the multi-level metric in Figure 2 as an example. Seeds in the same $M_E$ clusters must have the same function coverage. However, since $M_E$ is more sensitive than
As discussed in [45], \( \succ \) is a partial order, so it is possible that two metrics are not comparable. To solve this problem, we propose a weaker principle: given two non-comparable metrics, we should cluster a seed with the metric that will select fewer seeds before the one that will select more seeds.

2) Examples: Following the above principles, we propose two multi-level coverage metrics as examples. Both examples use three-level clustering that works well in our evaluation.

The top-level metric in both examples is \( C_F \), which measures the function coverage. The middle-level metric is edge coverage \( C_E \). Functions invoked are essential runtime features that are commonly used to characterize an execution, and edge coverage is widely used in fuzzers such as AFL and LIBFUZZER. Notably \( C_E \succ_s C_F \).

The most important one is the bottom-level metric, which is the most sensitive one. In this work, we mainly evaluated a bottom-level metric called distance metric \( C_D \). It traces conditional jumps (i.e., edges) of a program execution, calculates the hamming distances of the two arguments of the conditions as covered features, and treats each observed new distance of a conditional jump as new coverage. Unlike \( C_E \) or \( C_F \) that traces control flow features, \( C_D \) focuses on data-flow features and actively accumulates progress made in passing condition checks for fuzzing.

To understand whether our approach can support different coverage metrics (fitness functions), we also evaluated another coverage metric called memory access metric \( C_A \). As the name implies, this metric traces all memory reads and writes, captures continuous access addresses as array indices, and treats each new index of a memory access as new coverage. \( C_A \) pays attention to data flow features and accumulates progress made in accessing arrays that might be long. To distinguish memory accesses that happen at different program locations, the measurement also includes the address of the last branch. However, since not all basic blocks contain memory accesses, \( C_A \) is not directly comparable to \( C_E \) using sensitivity. However, we observe that \( C_A \) can generate much more seeds than \( C_E \), so \( C_A \) comes after \( C_E \) and its measurement \( M_A \) stays at the bottom level.

IV. HIERARCHICAL SEED SCHEDULING

This section discusses how to schedule seeds against hierarchical clusters generated by a multi-level coverage metric.

A. Scheduling against A Tree of Seeds

Conceptually, a multi-level coverage metric \( C_n \sim (C_1 \cdots C_n) \) organizes coverage measurements \( (M_i) \) and seeds as a tree, where each node at layer (or depth) \( i \in \{1, \ldots, n\} \) represents a cluster represented by \( M_i \) and its child nodes at layer \( i+1 \) represent sub-clusters represented by \( M_{i+1} \). At leaf-level, each node is associated with real seeds. Additionally, at layer 0 is a virtual root node representing the whole tree. To schedule a seed, the scheduler needs to seek a path from the root to a leaf node.

Exploration vs Exploitation. The main challenge a seed scheduler faces is the trade-off between seed exploration (trying out other fresh seeds) and exploitation (keep fuzzing a few interesting seeds to trigger a breakthrough). On the one hand, fresh seeds that have rarely been fuzzed may lead to surprisingly new coverage. On the other hand, a few valuable seeds that have led to significantly more new coverage than others in recent rounds encourage to focus on fuzzing them.

Organizing seeds in a tree with hierarchical clusters facilitates a more flexible control over the seed exploration and exploitation. Specifically, fuzzers can focus on a single cluster in which seeds cover the same functions at the first layer and then try out many (sub-)clusters with seeds exercising different edges at the second layer. Alternatively, fuzzers can also try out seeds exercising different groups of functions, then only pick seeds covering some specific edges.
In this work, we explore the feasibility of modeling the fuzzing process as a multi-armed bandit (MAB) problem [49], [54] and using the existing MAB algorithms to balance between exploitation and exploration. After trying several different MAB algorithms, we decide to adopt the UCB1 algorithm [2], [5] to schedule seeds, since it works the best empirically, despite being one of the simplest MAB algorithms. As illustrated by function SelectNextSeedToFuzz() in Algorithm 3, starting from the root node, our scheduling algorithm selects the child node with the highest score, which is calculated based on the coverage measurements, until reaching the last layer to select among leaf nodes that are associated with real seeds. Because all seeds have the same coverage at the leaf level, we scheduling them with round robin for simplicity.

At the end of each round of fuzzing, nodes along the scheduled path will be rewarded based on how much progress the current seed has made in this round, e.g., whether there are new coverage features exercised by all the generated test cases. In this way, seeds that perform well are expected to have increased scores for competing in the following rounds, while seeds making little progress will be de-prioritized.

Note that a traditional MAB problem assumes a fixed number of arms (nodes in our case) so that all arms can get an estimation of their rewards at the beginning. However, our setup breaks this assumption since the number of nodes grows as more and more seeds are generated. To address this issue, we introduce a rareness score of a node, so that each new node will have an initial score to differentiate itself from other new nodes. We will discuss seed scoring in more detail later in §IV-B.

It is also worth mentioning that a recent work Ecofuzz [54] proposed using a variant of the adversarial multi-armed bandit (AMAB) model to perform seed scheduling. However, it cannot solve the seed exploration problem caused by more sensitive coverage metrics, as it attempts to explore all existing seeds at least once. Moreover, we have also experimented with the EXP3 algorithm that aims to solve the AMAB problem, but it performed worse than UCB1 in our setup.

### B. Seed Scoring

How to score seeds directly affect the trade-off between exploration and exploitation. First, for exploitation, seeds that have performed well recently should have high scores as they are expected to make more progress. Second, for exploration, the scoring system should also consider the uncertainty of rarely explored seeds. We extended the UCB1 algorithm [2], [5] to achieve a balance between exploitation and exploration. From a high level, our scoring method considers three aspects of a seed: (1) its own rareness, (2) easiness to discover new seeds from this seed, and (3) uncertainty.

In order to discuss this in more detail, let us first define some terms more formally. First, we define the hit count of a feature $F \in T_l$ at level $l$ as the number of test cases ever generated that cover the feature.

**Definition IV.1.** Let $P$ be the program under fuzzing, $I$ be the set of all test cases that have been generated so far. The hit count of a feature $F$ is $\text{num}\_\text{hits}[F] = |\{I \in I : F \in C_l(P, I)\}|$.

As observed in [9], features that are rarely exercised by test cases deserve more attention because they are not likely to be exercised by valid inputs. The rareness of a feature describes how rarely it is hit, which is the inverse of the hit count.

**Definition IV.2.** The rareness of a feature $F$ is $\text{rareness}[F] = \frac{1}{\text{num}\_\text{hits}[F]}$.

Before describing how we calculate the reward of a round of fuzzing, we first define the feature coverage of fuzzing seed $s$ at round $t$.

**Definition IV.3.** Let $P$ be the program under fuzzing, $I_{s,t}$ be the set of test cases generated at round $t$ via fuzzing seed $s$. We denote the feature coverage at level $C_l$, $l \in \{1, \cdots , n\}$ as $f_{\text{cov}}[s, l, t] = \{F : F \in C_l(P, I) \forall I \in I_{s,t}\}$

Next, we describe how we calculate the reward to the seed just fuzzed after a round of fuzzing. An intuitive way is to count the number of new features covered as the reward. However, we quickly noticed that this does not work well. As the fuzzing campaign goes on, the probability of exercising new coverage is dramatically decreased, indicating that a seed can hardly obtain new rewards. Consequently, the mean reward of seeds may quickly decrease to zero. When we have many seeds with minor variances near zero, the UCB algorithm cannot properly prioritize seeds. Moreover, under the common observation that infrequent coverage features deserve more exploration than others, seeds that can lead to inputs that exercise rare features are definitely more valuable, even if they do not cover new features. Motivated by these observations, we take the rareness of the rarest feature that is exercised by all generated inputs as the reward to the schedule seed. Formally, for a seed $s$ that is fuzzed at round $t$, its fuzzing reward w.r.t. coverage metric $C_l$ is

$$\text{SeedReward}(s, l, t) = \max_{F \in f_{\text{cov}}[s, l, t]} \left(\text{rareness}[F]\right)$$

Based on the seed reward, we compute the reward to a
cluster by propagating seed rewards to clusters scheduled at upper levels. More formally, let \( \langle a^1, \ldots, a^n, a^{n+1} \rangle \) be the sequence of nodes (in the seed tree) selected at round \( t \), where \( a^{n+1} \) is the seed node for \( s \) and \( a_i \) is coverage measurements for the corresponding clusters. Since scheduling node \( a' \) affects the following scheduling of nodes \( a^{l+1}, \ldots, a^n \) at lower layers, the reward of node \( a' \) as feedback consists of the seed reward regarding coverage levels \( l, l+1, \ldots, n \) as illustrated in Equation 2. Note that we use the geometric mean here since it can handle different scalars of the involved values with ease.

\[
\text{Reward}(a^l, t) = \frac{n - l + 1}{\prod_{t \leq k \leq n} \text{SeedReward}(s, k, t)} \tag{2}
\]

Right now, we are able to estimate the expected performance of fuzzing a node using the formula of UCB1 \([2, 5]\). Formally, the fuzzing performance of a node \( a \) is estimated as

\[
\text{FuzzPerf}(a) = Q(a) + U(a) \tag{3}
\]

\( Q(a) \) is the empirical average of fuzzing rewards that \( a \) obtains so far, and \( U(a) \) is radius of the upper confidence interval.

Unlike UCB1 which calculates \( Q(a) \) using the arithmetic mean of the rewards that node \( a \) obtains so far, we use the weighted arithmetic mean instead. More specifically, during the fuzzing, the rareness of a feature is decreasing as it is exercised by more and more test cases. As a result, even the same fuzzing coverage can lead to different fuzzing rewards for mutating a seed: the reward of an earlier round might be significantly higher than that of a later round. To address this issue, we introduce a discount factor as weight in order to favor newer rewards rather than older ones. More formally, given a node \( a \) that is selected for round \( t \), we update its weighted mean at the end of round \( t \) in such a way that we progressively decrease the weight to the previous mean reward in order to give higher weights to newer rewards as illustrated in Equation 4.

\[
Q(a, t) = \frac{\text{Reward}(a, t) + w \times Q(a, t') \times \sum_{p=0}^{N[a, t]-1} w^p}{1 + w \times \sum_{p=0}^{N[a, t]-1} w^p} \tag{4}
\]

\( N[a, t] \) denotes the number of times that node \( a \) has been selected so far at the end of round \( t \), \( t' \) is the last round at which node \( a \) was selected, and \( w \) is the discount factor. Note that the smaller \( w \) is, the more we ignore the past rewards. When \( w \) is set to 0, all the past rewards are ignored. To study how \( w \) affects the fuzzing performance, we conduct an empirical experiments \([5V-F]\). Based on the results, we empirically set \( w \) to 0.5 in our evaluation.

\( U(a) \) is the estimated radius factoring in the number of times \( a \) has been selected. In addition, we also consider the number of seeds that \( a \) contains based on the insight that nodes with more seeds should be scheduled more for seed exploration. More formally, given a seed \( a \) and its parent \( a' \), we calculate \( U(a) \) as

\[
U(a) = C \times \sqrt{\frac{Y[a]}{Y[a']}} \times \frac{\log N[a']}{N[a]} \tag{5}
\]

\( Y[a] \) denotes the number of seeds in the cluster of node \( a \), and \( N[a] \) denotes the times \( a \) has been selected so far. \( C \) is a pre-defined parameter that configures the relative strength of exploration and exploitation. In particular, a larger \( C \) results in a relatively wider radius in Equation 3 which encourages exploring fresh nodes that have been fuzzed fewer times. This can help a fuzzer get out of code regions that are too hard to solve. On the contrary, a smaller \( C \) indicates that the empirical average of fuzzing rewards gets weighted more, thus promoting nodes that have recently led to good progress. As a result, the fuzzer will focus on these nodes and is expected to reach more new code coverage. To further demonstrate how it affects the fuzzing performance, we fuzz the CGC benchmark with different values of \( C \) and show the results in \([5V-F]\). Based on the results, we set \( C \) to 1.4 in our evaluation.

The fuzzing performance estimated by Equation 3 based on fuzzing coverage is limited by what can be observed. This limitation can impact seeds that have never been scheduled and seeds that exercise rare features themselves but usually lead to inputs that exercise high-frequency features (e.g., for a program with rigorous input syntax checks, random mutations usually lead to invalid paths, hence lowering the reward). To mitigate this limitation, when evaluating a seed, we also consider features that it exercises. Particularly, we calculate the rareness of a seed via aggregating the rareness of features that it covers. More formally, given a seed \( s \), its rareness regarding \( M_l, l \in \{1, \ldots, n\} \) is

\[
\text{SeedRareness}(s, l) = \frac{\sum_{F \in C_l(P, s)} \text{rareness}^2[F]}{|\{F : F \in C_l(P, s)\}|} \tag{6}
\]

Note that here we take quadratic mean rather than, e.g., arithmetic mean because it preserves more data diversity. The rareness of a node \( a' \) measured by \( M_l \) is completely decided by its child seeds as they share the same coverage regarding \( M_l \). Let \( \langle a^1, \ldots, a^n, a^{n+1} \rangle \) be the sequence of nodes selected at round \( t \), where \( a^{n+1} \) is the leaf node representing a real seed \( s \), then at the end of round \( t \) the rareness of node \( a' \) is updated as

\[
\text{Rareness}(a') = \text{SeedRareness}(s, l) \tag{7}
\]

Notably, we update the rareness score of seeds and nodes lazily for two reasons. First, it reduces the performance overhead. Second, it can lead to overestimating the rareness of a node that has not been fuzzed for a long time, so that seed is more likely to be scheduled.

In addition to updating the rareness of a node picked in the past round, we also calculate the rareness of each new node similarly. As discussed previously, this makes each new
node have an initial score to differentiate itself from other new nodes before its reward is estimated.

Finally, we have the score of a node \( a \) via multiplying its rarityness and estimated fuzzing performance together as shown in Equation 8. This score is the one used in Algorithm 3 to determine which nodes will be picked and which seed will be fuzzed next.

\[
    \text{Score}(a) = \text{Rarityness}(a) \times \text{FuzzPerf}(a) \tag{8}
\]

V. Evaluation

Our main hypothesis is that our multi-level coverage metric and hierarchical seed scheduling algorithm driven by the MAB model can achieve a good balance between exploitation and exploration, thus boosting the fuzzing performance. To validate our hypothesis, we implemented two prototypes A-FL-HIER and AFL++-HIER, one based on AFL [55] and the other based on AFL++ [17], and evaluated them on various benchmarks aiming to answer the following research questions.

- **RQ1** Can AFL-HIER/AFL++-HIER detect more bugs than the baseline?
- **RQ2** Can AFL-HIER/AFL++-HIER achieve higher coverage than the baseline?
- **RQ3** How much overhead does our technique impose on the fuzzing throughput?
- **RQ4** How well does our hierarchical seed scheduling mitigate the seed explosion problem caused by high sensitive of coverage metrics?
- **RQ5** How do the hyper-parameters affect the performance of our hierarchical seed scheduling algorithm?
- **RQ6** How flexible is our framework to integrate other coverage metrics?

A. Experiment Setup

1) Benchmarks: The first set of programs are from DARPA Cyber Grand Challenge (CGC) [10]. These programs are carefully crafted by security experts that embed different kinds of technical challenges (e.g., complex I/O protocols and input checksums) and vulnerabilities (e.g., buffer overlow, integer overflow, and use-after-free) to comprehensively evaluate automated vulnerability discovery techniques. There are 131 programs from CGC Qualifying Event (CQE) and 74 programs from CGC Final Event (CFE), a total of 205. CGC programs are designed to run on a special kernel with seven essential system calls so that competitors can focus on vulnerability discovery techniques. In order to run those programs within a normal Linux environment, we use QEMU to emulate the special system calls. Unfortunately, due to imperfect simulation, some CGC programs fail to be run correctly. We also cannot handle programs that consist of multiple binaries, which communicate with each other through pre-defined inter-process communication (IPC) channels. As a result, we can only successfully fuzz 180 CGC programs (or binaries, in other words). We fuzz each binary for two hours and repeat each experiment 10 times to mitigate the effects of randomness. Each fuzzing starts with a single seed “1\,2\,3\,4\,5\,6\,7\,8\,9\,0”. We chose this initial because the initial seed affects the fuzzing progress a lot: seeds that are too good may make most code covered at the beginning if the program is not complex, while poor ones may make the fuzzing get stuck before reaching the core code of the program. These two cases both will make the fuzzing reach the plateau early, and fail to show the performance differences between our approach and other fuzzers. The seed we chose showed a good capability to reveal performance differences between fuzzers.

The second benchmark set is the Google FuzzBench [21] that offers a standard set of tests for evaluating fuzzer performance. These tests are derived from real-world open-sourced projects (e.g., libxml, openssl, and freetype) that are widely used in file parsers, protocols, and font operations. For this dataset, we used the standard automation script to run the benchmarks, so each benchmark uses the seeds provided by Google.

2) Implementations: For evaluation over the CGC dataset, we used a prototype built on top of the code open-sourced by Wang et al. [45], which is based on AFL QEMU-mode, for its support for binary-only targets and its emulation of CGC system calls. For evaluation over the FuzzBench dataset, we used a prototype built upon the AFL++ project [17] (QEMU-mode only), for its support of persistent mode and higher fuzzing throughput.

3) Baseline Fuzzers: For AFL-based prototype, we choose three fuzzers as the baseline for comparison: the original AFL [55], AFLFAST [9], and AFL-FLAT [45]. AFL-FLAT is configured with edge sensitivity \( C_E \) and distance sensitivity \( C_D \) (see §III-D2 for more details), but uses the power scheduler from AFLFAST instead of our hierarchical scheduler. As discussed in §III-A, the performance of greybox fuzzing is mainly affected by four factors: seed selection, seed scheduling, mutation strategies, and fuzzing throughput. We made all fuzzers use the same mutation strategy to reflect the benefit of our approach, and ran the experiments ten times to minimize the impact of randomness [24]. We also ran all fuzzers in the QEMU mode so they can have similar fuzzing throughput, which also makes it easier to assess AFL-HIER’s performance overhead. Comparisons with AFL and AFLFAST aim to show the overall performance improvement of AFL-HIER; and comparison with AFL-FLAT aims to show the necessity/benefit of our scheduler (i.e., increasing the sensitivity of the coverage metric alone is not enough).

For AFL++-based prototype, we choose two fuzzers as the baseline: the original AFL++ [17] and AFL++-FLAT. We ran all fuzzers in the QEMU-mode and enabled persistent mode for better throughput.

4) Computing Resources: All the experiments are conducted on a 64-bit machine with 48 cores (2 Intel(R) Xeon(R) Platinum 8260 @2.40GHz), 375GB of RAM, and Ubuntu 18.04. Each fuzzing instance is bound to a core to avoid interference.

\footnote{We are working with Google to provide a more thorough comparison with other fuzzers.}

\footnote{The version is 2.68c which our prototype is built on.}
AFL-HIER crashes more binaries and faster. Especially, it crashes the vulnerability, this number equals the total number of unique trials. Note that since each binary supposedly only has one vulnerability, this number equals the total number of unique crashes in 2 hours.

In this experiment, we evaluate fuzzers’ capability of detecting known bugs embedded in the CGC binaries. Figure 3a shows the number of crashed CGC binaries across ten rounds of trials. Note that since each binary supposedly only has one vulnerability, this number equals the total number of unique crashes. On average, AFL crashed 64 binaries, AFL-FAST crashed 61 binaries, and AFL-FLAT crashed 62 binaries. In contrast, AFL-HIER crashes about 77 binaries on average, which is about 20% more binaries in the 2-hour fuzzing campaign. AFL-HIER also performed much better when we look at the lower and upper bound: its lower bound of crashes (74) is always higher than the upper bound of all other fuzzers. Notably, these vulnerabilities are carefully designed by security experts to highly mimic real-world security-critical vulnerabilities.

Table I shows the pairwise comparisons of uniquely crashed binaries on CGC benchmark.

### Table I: Pairwise comparisons (row vs. column) of uniquely crashed on CGC benchmark.

<table>
<thead>
<tr>
<th></th>
<th>AFL</th>
<th>AFLFAST</th>
<th>AFL-FLAT</th>
<th>AFL-HIER</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>-</td>
<td>8</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>AFLFAST</td>
<td>3</td>
<td>-</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>AFL-FLAT</td>
<td>11</td>
<td>13</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>AFL-HIER</td>
<td>17</td>
<td>22</td>
<td>18</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 3: Crash detection on CGC benchmarks.

B. RQ 1. Bug Detection

* In experiments with CGC benchmarks, AFL-HIER crashes more binaries and faster. Especially, it crashes the same number of binaries in 30 minutes, that AFL-Fast crashes in 2 hours.

In this experiment, we evaluate fuzzers’ capability of detecting known bugs embedded in the CGC binaries. Figure 3a shows the number of crashed CGC binaries across ten rounds of trials. Note that since each binary supposedly only has one vulnerability, this number equals the total number of unique crashes. On average, AFL crashed 64 binaries, AFL-FAST crashed 61 binaries, and AFL-FLAT crashed 62 binaries. In contrast, AFL-HIER crashes about 77 binaries on average, which is about 20% more binaries in the 2-hour fuzzing campaign. AFL-HIER also performed much better when we look at the lower and upper bound: its lower bound of crashes (74) is always higher than the upper bound of all other fuzzers. Notably, these vulnerabilities are carefully designed by security experts to highly mimic real-world security-critical vulnerabilities.

**Table I** shows the pairwise comparisons of uniquely crashed on CGC benchmark.

C. RQ 2. Code Coverage

* Results on CGC binaries demonstrate that AFL-HIER generally achieved more code coverage and achieved the same coverage faster. Specifically, AFL-HIER increases the coverage by more than 100% for 20 binaries, and achieves the same coverage in 15 minutes that AFL-Fast achieves in 120 minutes for about half of the binaries. On FuzzBench, AFL++-HIER achieved higher coverage on 10 out of 20 projects.

CGC Benchmark. In this experiment, we first measured the edge coverage achieved by fuzzers using QEMU (i.e., captured during binary translation) on CGC binaries.

Figure 4a illustrates the mean code coverage increase of AFL-HIER over other fuzzers for the 180 CGC binaries, after 2 hours of fuzzing. The curve above 0% means AFL-HIER covered more and the curve below 0% means AFL-HIER covered less. The x-axis presents the accumulated number of binaries within a 95% confidence, and the y-axis shows the increased coverage in logarithmic scale. For example, there are about 20 binaries for which the code coverage is increased by at least 100%, and about 45 binaries for which the code coverage is increased by at least 10%. After 2 hours of fuzzing, AFL-HIER achieved more coverage for about 90 binaries than other fuzzers and achieved the same coverage for 50 binaries. Among about 30 binaries on which AFL-HIER achieves less coverage, on half of them the difference is lower than 2%; and only on five of them the difference is greater than 10%. This result shows that our hierarchical scheduler not only can find many unique bugs but also can efficiently explore the search space.
D. RQ 3. Fuzzing Throughput

*Results on CGC benchmarks show that AFL-HIER has a competitive throughput as AFL and AFLFAST. Moreover, even built on the faster fuzzer AFL++, AFL++-HIER still*

3We are working with Google to provide a 23-hour run that compares with more fuzzers.
has a comparable throughput as shown by the results on FuzzBench benchmarks.

A multi-level coverage metric requires collecting more coverage measurements during runtime and performing more operations to insert a seed into the seed tree. Similarly, our hierarchical scheduler also requires more steps than the power scheduler of AFL and AFLFAST. Therefore, we expect our approach to have a negative impact on fuzzing throughput. Moreover, the multi-level coverage metric is sensitive to minor variances of test cases and execution paths; consequently, it is more likely to schedule larger and more complex seeds leading to longer execution time.

To quantify the impact on fuzzing throughput, we first investigated the proportion of the time that AFL-HIER spends in scheduling, which involves maintaining the incidence frequencies and the tree of seeds and choosing the next seed to fuzz. The results on CGC benchmarks are shown in Figure 7, where the x-axis represents individual runs (in total $10 \times 180 = 1800$) and the y-axis shows the portion of time spent on scheduling. We can see that the median overhead is as low as 3%, and most overhead is lower than 10%. On AFL++-based prototype, we observed lower performance overhead, as shown in Figure 9.

Next, we measured the throughput of AFL-HIER versus AFL and AFLFAST on CGC benchmarks. Figure 6 shows the ratio of AFL-HIER’s throughput over AFL and AFLFAST in an ascending order. The x-axis represents different CGC binaries while the y-axis shows the ratio within a 95% confidence in logarithmic scale. Surprisingly, AFL-HIER only leads to a lower throughput for about a quarter of the binaries; and for another quarter of the binaries, AFL-HIER’s throughput is at least twice as AFLFAST’s. This indicates that the specific optimizations for AFL-HIER act very well. A similar trend is also observed on the AFL++-based prototype, as shown in Figure 8.

E. RQ 4. Performance Boost via Hierarchical Seed Scheduling

* Experiment results on CGC and FuzzBench benchmarks demonstrate that our hierarchical seed scheduler dramatically reduces the number of candidates to be examined.

Previous experiments already show that our hierarchical seed scheduler is more suitable for highly sensitive coverage metrics, as AFL-HIER can achieve higher coverage faster than AFL-FLAT and find more bugs. In this evaluation, we
TABLE II: Unique edge coverage between afl++ (Qemu) and afl++-hier (Hier) on FuzzBench benchmarks. The coverage is union over different runs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>Hier - Qemu</th>
<th>Qemu - Hier</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloaty_fuzz_target</td>
<td>102417</td>
<td>24</td>
<td>674</td>
</tr>
<tr>
<td>curl_curl_fuzzer_http</td>
<td>143182</td>
<td>203</td>
<td>114</td>
</tr>
<tr>
<td>freetype2-2017</td>
<td>56114</td>
<td>774</td>
<td>1227</td>
</tr>
<tr>
<td>harfbuzz-1.3.2</td>
<td>13073</td>
<td>58</td>
<td>124</td>
</tr>
<tr>
<td>jsoncpp_jsoncpp_fuzzer</td>
<td>2583</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libm-2017-03-21</td>
<td>12817</td>
<td>36</td>
<td>33</td>
</tr>
<tr>
<td>libjpeg-turbo-07-2017</td>
<td>18486</td>
<td>0</td>
<td>237</td>
</tr>
<tr>
<td>libpng-1.2.56</td>
<td>5944</td>
<td>6</td>
<td>54</td>
</tr>
<tr>
<td>libxml2-v2.9.2</td>
<td>89852</td>
<td>52</td>
<td>210</td>
</tr>
<tr>
<td>mbedtls_fuzz_dtlsclient</td>
<td>32046</td>
<td>142</td>
<td>102</td>
</tr>
<tr>
<td>openssl_x509</td>
<td>115381</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>openthread-2019-12-23</td>
<td>42901</td>
<td>344</td>
<td>0</td>
</tr>
<tr>
<td>proj4-2017-08-14</td>
<td>10434</td>
<td>109</td>
<td>67</td>
</tr>
<tr>
<td>re2-2014-12-09</td>
<td>5904</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>sqlite3_osfuzz</td>
<td>48181</td>
<td>1880</td>
<td>965</td>
</tr>
<tr>
<td>systemd_fuzz-link-parser</td>
<td>4167</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vorbis-2017-12-11</td>
<td>6372</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>woff2-2016-05-06</td>
<td>6401</td>
<td>54</td>
<td>8</td>
</tr>
<tr>
<td>zlib_zlib_uncompress</td>
<td>1664</td>
<td>24</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 6: Comparison between Throughput of AFL-HIER, AFL, AFLFAST and AFL-FLAT on CGC benchmarks.

investigate the number of seeds generated by each fuzzer to validate that such improvement is indeed caused by the scheduler. [Figure 10] shows the number of seeds generated by each fuzzer on the left side, as well as the number of nodes at different levels of the tree in AFL-HIER on the right side. The y-axis is in logarithmic scale. We can observe that due to the increased sensitivity of distance metric $C_D$, both AFL-HIER and AFL-FLAT selected one magnitude more seeds than AFL and AFLFAST, which uses edge coverage with hit count. However, by clustering the seeds in a hierarchical structure, AFL-HIER dramatically reduced the number of candidates to examine when scheduling. Specifically, on average there are about $21 + 1102/21 + 2350/1102 + 2608/2350 \approx 77$ examinations to perform for each scheduling, which is significantly less than examining 2608 seeds. As a result, even with the most number of seeds (more than AFL-FLAT), AFL-HIER can still balance exploration and exploitation and achieve better fusing performance (in terms of coverage and detected bugs) than baseline fuzzers.

On FuzzBench benchmarks, we also observed a similar level of reduction, as shown in [Figure 11] More importantly, we can see that our scheduling algorithm can scale to larger programs with significantly more edges and more saved seeds. As shown in Table II, all the benchmarks have at least thousands of edges in total, and some even contain more than one hundred thousand edges.

F. RQ 5. Hyper-parameters

* Experiment results on CGC benchmarks demonstrate that the hyper-parameters will affect the performance in terms of crashes and edge coverage.

As discussed in §IV-B, the seed scoring involves two hyperparameters. One is $w$ in Equation 4 that determines how much
we will decrease weights to old rewards when calculating the mean reward. The other one is \( C \) in Equation 5 that controls the trade-off between seed exploration and exploitation. In this evaluation, we investigate when they are set to different values, how the fuzzing performance will vary. Table III and Table V show the average number of crashed binaries and covered edges with different values of \( C \) and \( w \), respectively. In addition, we also investigate the number of binaries uniquely crashed as shown in Table IV and Table VI, where each cell represents the number of binaries that have been crashed by the setting of the row once but never by the setting of the column. We can observe that different settings will lead to different results.

Notably, when \( C \) is set to 0, which extremely encourages exploitation, it uniquely crashes the most binaries, but on average, it crashes the least. This indicates that although keeping exploitation may help to trigger a crash at the end of a seed chain in one run, it also takes the risk of being trapped in fuzzing other seeds that previously have led to rarely explored coverage, thus missing the crash in other runs. In other words, high exploitation may do better in crash triggering than crash reproducing. Meanwhile, the result of edge coverage indicates that exploring more coverage may not be closely related to bug detection as expected when under different configurations of the relative strength of exploration and exploitation. For example, setting \( C \) to 0.014 will lead to significantly less coverage, but it crashes almost the same number of binaries as others.

In terms of the hyper-parameter \( w \), note that a larger \( w \) makes old rewards more weighted, thus encourages seed exploitation rather than exploration. We can observe that setting \( w \) either too high (as 1.0) or too low (as 0.5) will lead to worse coverage, while setting \( w \) to 0.5 will lead to significantly more unique crashes.

Overall, we can observe that when setting \( C \) to 1.4, \( w \) to 0.5, they perform reasonably well in average crashes, unique crashes, and mean edge coverage. Thus we adapt these settings in our current implementation.

G. RQ 6. Ability to Support other Coverage Metrics

* Experiment results on the maze problem show that our hierarchical scheduler can also improve the fuzzing performance when using other sensitive coverage metrics.

As discussed in §III-A, it is very hard, if not impossible, to use edge or even distance coverage to solve the maze problem.
TABLE IV: Pairwise comparisons (row vs. column) of uniquely crashed on CGC benchmarks with different values of hyper-parameter C.

<table>
<thead>
<tr>
<th>Value of W</th>
<th>0.10</th>
<th>0.25</th>
<th>0.50</th>
<th>0.75</th>
<th>0.90</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>74</td>
<td>73</td>
<td>76</td>
<td>72</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Edge Cov</td>
<td>698</td>
<td>758</td>
<td>727</td>
<td>666</td>
<td>739</td>
<td>660</td>
</tr>
</tbody>
</table>

TABLE V: Average number of crashed CGC binaries and mean edge coverage with different values of hyper-parameter W.

<table>
<thead>
<tr>
<th>Value of W</th>
<th>0.10</th>
<th>0.25</th>
<th>0.50</th>
<th>0.75</th>
<th>0.90</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>74</td>
<td>73</td>
<td>76</td>
<td>72</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Edge Cov</td>
<td>698</td>
<td>758</td>
<td>727</td>
<td>666</td>
<td>739</td>
<td>660</td>
</tr>
</tbody>
</table>

(Listing 1). However, it is possible to solve it using memory sensitivity $C_A$ (see §III-D2 for details). In this experiment, we investigate whether our hierarchical scheduler can also boost the performance of coverage metrics other than code-related coverage. Specifically, we configured AFL-FLAT and AFL-HIER to use memory access metric $C_A$ instead of distance metric $C_D$ and evaluate the two fuzzers on the maze problem. Table VII shows the results. As we can see, compared to the power scheduler used by AFL-FLAT, our hierarchical scheduler allows AFL-HIER to solve the maze problem much faster. This empirical result suggests that our scheduler is flexible to support different coverage metrics.

TABLE VII: Average solving time for the maze problem of hyper-parameter W.

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>AFL-FLAT</th>
<th>AFL-HIER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>383 ± 92</td>
<td>180 ± 36</td>
</tr>
</tbody>
</table>

VI. RELATED WORK

A. Coverage Guided Greybox Fuzzing

Greybox fuzzing was introduced as early as in 2016 by Sidewinder [16]. Since then it has been extensively used in practice with the popularity of AFL [55] and LIBFUZZER [1]. Meanwhile, it has gained tremendous academic interest in various areas. On the one hand, various techniques including taint tracking [12], symbolic execution [4], program transformation [23], deep learning [30], [40], are incorporated into greybox fuzzing to boost its performance.

Some research focuses on finding domain-specific bugs via specially designed coverage metrics. MemLock [38] takes memory consumption into account when evaluating a test case in order to trigger memory consumption bugs. SlowFuzz [34] counts the number of instructions executed by test case as coverage features to detect algorithm complexity bugs. Furthermore, PerfFuzz [26] records the number of times each block is executed by a test case and considers the test case as a new seed if it increases the execution count for any block in order to find hot spots. KRACE [50] develops a new coverage that captures the exploration progress in the concurrency dimension to find data races in kernel file systems. Our work offers a framework to combine these metrics with others so that they can benefit from more general metrics.

Wang et al. [45] systematically evaluate multiple coverage metrics, revealing that there is no grand slam coverage metric that can beat others, and it is promising to combine different coverage metrics together through cross seeding between multiple fuzzing instances. We combine coverage metrics within one fuzzing instance, avoiding the overhead of synchronizing seeds as well as redundant fuzzing. FuzzFactory [12] provides a platform that makes combining different coverage metrics easy and flexible. However, it does not address the seed explosion problem, as our experimental results demonstrate.

B. Improving Coverage Metric

Angora [12] involves calling context, and MemFuzz [15] involves memory accesses when calculating edge coverage to explore program states more pervasively. However, they pay little attention to the potential seed explosion problem. CollAFL [19] improves edge coverage accuracy via ensuring each edge has a unique hash, and utilizes various kinds of coverage information to prioritize seeds. However, it requires a precise analysis of the control flow graph of the target program. Wang et al. [47] differentiate edges based on their corresponding memory operations for seed prioritization to find memory corruption bugs. However, it is incapable of preventing a test case involving diverse memory operations from being dropped since it still relies on edge coverage to evaluate the quality of test cases. Greyone [18] augments edge coverage with data flow features where lightweight and accurate taint tracking is necessary. IJON [3] designs various primitives for annotating source code that will adapt the coverage metric to different kinds of challenges of exploring deep state space. However, it requires domain knowledge of the target program and much manual work. Ankou [30] proposes a new coverage metric that measures distances between execution paths of test cases, and employs adaptive seed pool update to mitigate seed explosion. By comparison, the distance we propose is between two arguments of conditions in conditional branches, and we address the seed explosion problem via organizing the seed pool as a multi-level tree.

Our approach relies little on these techniques and is orthogonal to these work. On the other hand, there is an increasing number of greybox fuzzers that are carefully crafted to test specific types of programs such as OS kernels [38], [53], firmware [57], protocol [55], smart contracts [31], deep neural networks [50]. It is promising for these fuzzers to adopt our techniques to improve their efficiency.
that randomly combining different metrics without a proper organization may lead to negative impacts.

C. Smart Seed Scheduling

AFLFAST [9] focuses on fuzzing seeds exercising low-frequency paths and assigns more power to them through modeling greybox fuzzing as a Markov chain. FairFuzz [27] identifies low-frequency edges and prioritizes mutations satisfying these edges. Entropic [2] targets on the test cases that a seed has generated, evaluating the diversity of coverage features they exercise via the information-theoretic entropy. Consequently, seeds with higher information gains are more likely to be scheduled. Vuzzer [27] de-prioritizes seeds hitting error-handling or frequently visited code that is identified via heavyweight static and dynamic analysis. Cerebo [28] prioritizes seeds via various metrics including code complexity, coverage, and execution time. AFLGo [8] and UAFL [44] are directed fuzzers that favor seeds closer to targeted code. Compared to these work, our scheduling algorithm considers the rareness of both static features it covers and test cases it has generated when evaluating a seed.

Modeling scheduling as an MAB problem. Woo et al. [49] model blackbox mutational fuzzing as a classic Multi-Armed Bandit (MAB) problem. Nevertheless, its goal is to search for an optimal arrangement for a fixed set of program-seed pairs to maximize the unique bugs found. EcoFuzz [54] proposes a variant of the Adversarial Multi-Armed Bandit model for modeling seed scheduling. However, it explicitly puts seed exploration and exploitation in separate stages, launching exploitation only when all existing seeds have been explored once. Thus it is incapable of solving the seed explosion problem.

VII. CONCLUSION

Fine-grained coverage metrics, such as distances between operands of comparison operations and array indices involved in memory accesses, allow greybox fuzzers to detect bugs that cannot be triggered by traditional edge coverage. However, existing seed scheduling algorithms cannot efficiently handle the increased number of seeds. In this work, we present a new coverage metric design called multi-level coverage metric, where we cluster seeds selected by fine-grained metrics using coarse-grained metrics. Combined with a reinforcement-learning-based hierarchical scheduler, our approach significantly outperforms existing edge-coverage-based fuzzers on DARPA CGC challenges.

ACKNOWLEDGMENT

This work is supported, in part, by National Science Foundation under Grant No. 1664315, No. 1718997, Office of Naval Research under Award No. N00014-17-1-2893, and UCOP under Grant LPR-18-548175. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.