Towards Hybrid Fuzzing with Multi-level Coverage Tree and Reinforcement Learning in Greybox Fuzzing

Dixi Yao
University of Toronto
Toronto, Canada
dixi.yao@mail.utoronto.ca

Kai Shen
University of Toronto
Toronto, Canada
kai.shen@mail.utoronto.ca

Xiaochong Wei
University of Toronto
Toronto, Canada
xiaochong.wei@mail.utoronto.ca

ABSTRACT

Coverage-guided greybox fuzzing is considered the state-of-the-art testing technique in vulnerability detection. As for the promising vulnerability detection technique, we address that there are still two challenges to be resolved for better performance. On the one hand, code coverage metrics should be more informative to distinguish between various program executions. On the other hand, seed scoring algorithms should be well-designed for a better balance between seed exploration and seed exploitation.

In this paper, we propose a two-fold hybrid solution to tackle these unresolved challenges. We leverage a multi-level coverage tree to take advantage of various coverage metrics with efficiency. Meanwhile, state-of-the-art reinforcement learning algorithms are being leveraged for intelligently scoring the seeds. Evaluations of our work are conducted on a lightweight JSON parser benchmark, which is implemented for experiments under low computation budgets. More importantly, it reveals the superiority of our approach on the basis of unique crashes, unique hangs, and total covered paths.

KEYWORDS

Coverage-guided greybox fuzzing, multi-level coverage metrics, reinforcement learning

1 INTRODUCTION

Coverage-guided greybox fuzzing [18] is considered as the state-of-the-art testing technique in vulnerability detection. It has been widely adopted in realistic software vulnerability detections to successfully find tens of thousands of bugs. Different from traditional fuzzing paradigms, coverage-guided greybox fuzzing, empowered by real-world fuzzing observations, was proposed and has received a tremendous amount of research attention from academia and industry alike. For instance, American Fuzzy Loop (AFL) [29] is responsible for the discovery of hundreds of notable vulnerabilities and other interesting bugs in many applications [11], has helped make countless non-security improvements to core tools, and has a large community of security researchers involved in extending it.

As an emerging fuzzing solution, the essence of the coverage-guided greybox fuzzing design philosophy is distinguished mainly in two aspects. On the one hand, it tracks code coverage information and utilizes it to guide future fuzzing. On the other hand, it leverages lightweight instrumentation techniques to determine a unique identifier for the path that is exercised by an input with negligible computation overhead [1]. Since the proposed coverage-guided greybox fuzzing has evolved for many years and it is easy for deployment, it has become the mainstream with both satisfactory effectiveness and high efficiency.

While coverage-guided greybox fuzzing is a promising vulnerability detection technique for most popular software applications, there is still a lot of room for optimization that remains to be resolved even in state-of-the-art frameworks. As such, existing coverage-guided greybox fuzzing faced the following two challenges. Firstly, the coverage metrics used by current coverage-guided fuzzers should be more informative to distinguish different program executions as long as those executions achieve the same coverage under the given metric. That is, coverage inaccuracy could cause fuzzers to fail to differentiate between two different program paths in some cases, which may further lead to the neglect of potential vulnerabilities. Specifically, when a test case is exercising a new path that collides with a previously explored path, the coverage-guided fuzzer will wrongly classify the path as not interesting, then miss the chance to examine the test the path or its closely related path. As a result, inaccurate coverage information will cause a potential loss while its goal is to find a complete collection of vulnerabilities. In short, we argue that most of the current fuzzers usually only considers a onefold code coverage metric, which may be coarse-grained or fine-grained, and it can not avoid from the performance suffering because of some critical information loss.

The second challenge is the dilemma of the trade-off between seed exploration and seed exploitation. It is worth noting that in greybox fuzzing, the task of seed exploration aims to select as many new seeds as possible, while seed exploitation targets on sticking fuzzing a few interesting seeds rather than preferring new seeds. Therefore, on the one hand, fresh seeds that has rarely been fuzzed may lead

to surprisingly new coverage [3]. 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 [26]. Hence, traditional hand-crafted seed scoring algorithms are not intelligent enough to meet the need for large amounts of scenarios for satisfactory generalization capability, and more intelligent and time-varying seed scoring algorithms have become an urgent need.

In this paper, we present a promising solution to solve the existing challenges in the area of coverage-guided greybox fuzzing. We aim to address two parts of the improvements as a hybrid solution to solve the difficulties separately. For the first part, we leverage a more fine-grained multi-level coverage tree so as to allow the fuzzer to detect bugs that cannot be covered by traditional coverage metrics. It is worth noting that the multi-level coverage tree can be constructed with flexibility to better accomodate different vulnerability detecting scenes. For the second part, we apply reinforcement learning algorithms to replace traditional hand-crafted seed scoring policies. Thanks to the great learning ability of state-of-the-art reinforcement learning algorithms, it can outperform existing seed scoring algorithms to achieve a better balance between seed exploration and seed exploitation.

Contributions are listed as follows:

- We leverage a more fine-grained multi-level coverage tree as the code coverage metric, which makes the coverage metrics more informative to be able to avoid missing potential vulnerabilities.
- We utilize state-of-the-art reinforcement learning algorithms to replace the traditional seed scoring policies, which empowers the fuzzer the achieve a better trade-off between seed exploration and seed exploitation.
- To better evaluate our proposed solution within limited computation budget, we implement a simple and lightweight JSON parser benchmark.
- We evaluated our prototypes with different parameter settings based on our benchmark. The results show the effectiveness and superiority of the hybrid framework of both multi-level coverage trees and reinforcement-learning-based seed scoring algorithms.

The remaining of this paper is organized as follows. Section 2 demonstrates the background and related work of coverage-guided greybox fuzzing. Section 3 illustrates the system design of our solution, which includes overview, multi-level coverage tree, and seed score policy based on reinforcement learnings. Section 4 details explains our evaluation, including the implemented lightweight benchmark and the conducted experiments. Section 5 concludes our learned lessons from ECE 1776, Computer Security, Cryptography and Privacy. Lastly, we present the conclusion of this paper in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 Fuzzing

Nowadays, fuzzing [13, 14] has become the most effective and efficient state-of-the-art vulnerability discovery solution [7]. In programming, fuzzing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a program for testing. Meanwhile, the program is under monitoring for exceptions such as crashes or memory leaks. More specifically, fuzzers will schedule and mutate a huge amount of testcases, then feed them to the target application for examination while the application's runtime execution states are being monitored for reporting the potentially detected bugs.

In general, fuzzers can be categoried in two types in terms of their test case generation approaches: grammar-based fuzzer too long [8, 10, 25] and mutation-based fuzzer [7, 24, 26]. The former leverages a known input grammar to generate test cases. However, it is not a mainstream approach as it requires much handcraft work for the translation of input grammars, and the more severe problem lies in the fact that it can not achieve its goal without the given grammar. On the contrary, mutation-based fuzzers can mutate testcases based on the existing seeds without the dependency on a pre-defined grammar. Hence, it is widely adopted in realistic fuzzing campaigns with simplicity and scalability. However, most of the mutation-based fuzzers are restricted by their poor code coverage informations [15].

2.2 Coverage-guided Fuzzing

To solve the poor coverage information problem of mutation-based fuzzing, one popular solution has been proposed and been proved effectively in finding many serious bugs in real software, which called coverage-guided fuzzing [16, 23]. In coverage-guided fuzzing, a fuzzing process maintains an input corpus containing inputs to the program under consideration [17]. Meanwhile, random changes are made to those inputs according to some mutation procedure, and mutated inputs are kept in the corpus when they exercise new coverages. In other words, this state-of-the-art solution achieves success by employing an evolving algorithm to drive fuzzers towards a high code coverage. Moreover, the most popular coverage-guided fuzzers for computer programs includes AFL [29], libFuzzer [22], and VUzzer [20].

As coverage metrics act as an important role in the fuzzing process, we investigated some popular coverage metrics. It is pointed out that when a coverage metric measures more fine-grained coverage information (e.g., edge coverage), it can dim the coarse-grained diversity (e.g., block coverage) among different seeds. However, a more fine-grained coverage of information will lead to more sensitivity, which means deeper exploration along the current path rather than

trying new paths. Thus, a ultimate goal is to carefully craft the seed scheduler to strike a balance between exploration and explication [30]. For instance, Bohme et al. [2] model the minimum effort to discover a neighboring seed as the required computation power, which can successfully use less power to make progress under a more sensitive coverage metric. However, each seed only carries a small step of progress, the accumulation of them can narrow the search space even faster.

2.3 Greybox Fuzzing

In addition to the aforementioned fuzzer classification, we also distinguish fuzzing styles based on another dimension, which is the degree of program analysis. There are three categories under this classification standard. The first one is black-box fuzzing [6, 28] which regards the program as a black box and only requires it to execute. The second type of fuzzing is white-box fuzzing [3, 5, 9, 24], which is based on symbolic execution and requires heavy-weight program analysis without the guarantee of unconstrained problem solving within a limited regime.

Greybox fuzzing [4, 19] is placed in-between the black-box fuzzing and the white-box fuzzing, and it only requires light-weight instrumentation to glean some program structure [1]. It outperforms black-box fuzzing with higher effectiveness because of more information about the internal structures, and outperforms white-box fuzzing with higher efficiency as a result of the reduction of the time-consuming heavy-weight program analysis. More specifically, greybox fuzzing acts as a trade-off between black-box fuzzing and white-box fuzzing, and the most distinctive step of greybox fuzzing is that, when executing a newly generated input, the fuzzer applies light-weight instrumentations to capture runtime features and expose them for further quality scoring of a generated test cases [26].

In more details, given a program to fuzz as well as an initial set of seeds, the greybox fuzzer will conduct a fuzzing process, which consists of a number of loops. In each loop, the fuzzer begins by selecting the following seed for fuzzing from the pool according to the scheduling criteria. Next, the scheduled seed will be assigned to a setted amount of computation budget and generate test cases as much as possible without exceeding the budget of this round. Lastly, test cases will be generated through mutation and crossover based on the scheduled seed [26].

2.4 Coverage-guided Greybox Fuzzing

As the classification methods mentioned above are orthogonal, coverage-guided greybox fuzzing has become the mainstream solution. It is an evolutionary process that maintains and evolves a population of test cases with the help of a

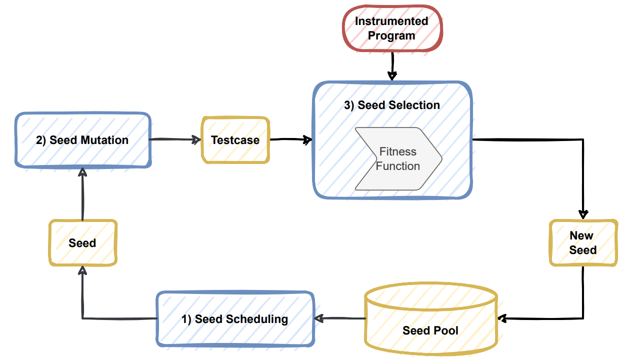


Figure 1: The Workflow of Coverage-guided Greybox Fuzzing

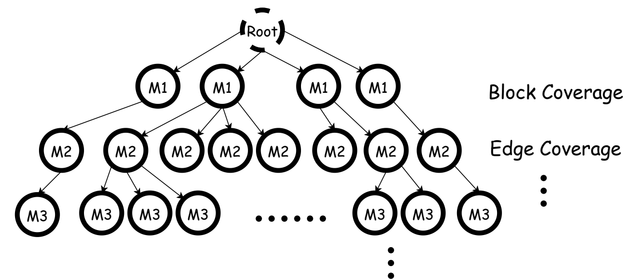


Figure 2: The multi-level coverage tree with three levels of measurements

fitness function, where the fitness function is used for deciding the quality of a given test case. Famous coverage-guided greybox fuzzer like AFL [29] and LibFuzzer [22] realized light-weight instrumentation to gain coverage information. For instance, AFL captures basic block transitions along coarse branch-taken hit counts in its instrumentation module. Furthermore, coverage-guided greybox fuzzing utilizes the coverage information to decide which generated inputs to retain for fuzzing and which input to fuzz next and for how long.

As illustrated in Figure 1, the workflow of the coverage-guided greybox fuzzing has three main stages, and it generates seeds incrementally via the demonstrated feedback loop. The first stage is named seed scheduling, where a seed will be picked from a set of seeds according to the given scheduling criteria. Next, a seed mutation stage will be performed. In the second stage, new test cases will be generated by performing various mutations on the scheduled seed within a limited time budget. The last stage is named as seed selection, where each generated test case will be fed to the target program under test and evaluated based on the pre-defined coverage metric. It is worth noting that if the generated test case leads to a new coverage, it will be selected as a new seed.

Hence, as the feedback loop continues, more coverage will be reached, and hopefully, a test case will trigger a bug. Although coverage-guided greybox fuzzing has drawn plenty of interests from both academia and industry, we argue that current fitness function still need to be more informative so as to distinguish more different executions, especially when some execution paths achieve the same coverage.

3 SYSTEM DESIGN

3.1 Overview

To allow a fuzzer to detect bugs that cannot be covered by traditional coverage metrics and have a better tradeoff between exploration and exploitation. We produced two improvements over the basic AFL++: Integrating a multi-level coverage tree and improving the seed score policy. We will first introduce the overview design of multi-level coverage tree [26] and then devel deeper into a very important factor which can strongly impact the effectiveness of such tree, the seed-score policy and discuss how to improve such policy with our novel reinforcement learning-based method.

3.2 Multi-level Coverage Tree

First, to detect more bugs, we leverage a multi-level coverage tree. In the multi-level coverage tree, there are many levels, which map to different measurements of coverage metrics. Each level can represent a grain level of coverage for example, the block coverage, the edge coverage and the distance metric coverage. Each time we generate a new test case based on the current seed, we will put the seed into the coverage tree based on the coverage. If a test case is assessed as exercising a new coverage by any of the measurements, it will be retained as a new seed and put in a proper cluster. The top-level measurement directly classifies all seeds into different clusters. Lower-level measurements work on each of the clusters generated by its parent node separately, classifying the seeds in it into smaller sub-clusters. Figure 2 shows a multi-level coverage tree with three levels.

For example, as shown in the figure 3. In each new run of instrument with a test case. First, we find if there is new block coverage and then check which block coverage it belongs to and put it on the subtree. Then we check level two of edge coverage and finally, we check which distance-based metric coverage it belongs to. By referring to distance-based metric coverage, it means the length between the last entry point and the current jumping edge. Then we place this seed on one of the leaf nodes. By the next time we want to exploit, we can select a seed from this tree based on current requirements.

3.3 Seed Score Policy

As we find during the clustering phase, it is important to correctly evaluate the seed. The good seed can lead to better

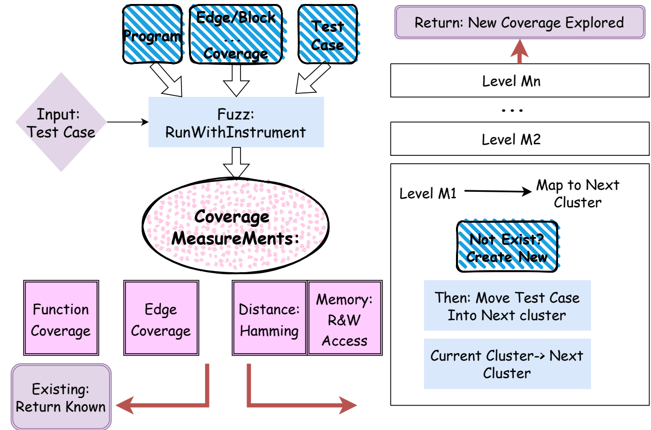


Figure 3: The flowchart of using seed clustering to do seed selection

cases where we can find more unique bugs. As a result, to evaluate whether a seed is good enough, we usually have a score policy. Then the fuzzer will be more prone to choosing those seeds with higher scores to generate new test cases during exploitation. In this section, we will first introduce the initial design of seed policy in AFL++ and then introduce our novel seed policy.

3.3.1 Preliminaries. In the original design of AFL++, it used mainly four metrics to evaluate the score of the seed:

- exec_us: The execution time of each test case.
- bitmap_size: The bitmap size of the test case.
- handicap: Handicap is proportional to how late in the game we learned about this path.
- depth: The depth of the current path, under the assumption that fuzzing deeper test cases is more likely to reveal stuff that can't be discovered with traditional fuzzers.

Then it will first set the default score value to 100 and then manually using many `if` conditions to modify the score on basis of each metric. For example, as shown in the example code 1, it will first compare the execution time with the average execution of all test cases, and change the perf score on basis of their ratio. However, such a setting lacks flexibility as there are a lot of manual settings. We need to first set several compare zones and then set how much we need to modify the performance score on the basis of human experience. First, by manually setting these hyperparameters, it is very hard for us to find the best hyperparameters. What's more, even if the set of hyperparameters is the best for the current fuzzed program, there is no guarantee it will fit to another program. For another program, we many need to change another set of hyperparameters.

Listing 1: Original design of function calculate_score().

```
u32 calculate_score(afl_state_t *afl, struct
    ↪ queue_entry *q) {

    u32 avg_exec_us = afl->total_cal_us / afl->
        ↪ total_cal_cycles;
    u32 avg_bitmap_size = afl->total_bitmap_size
        ↪  / afl->total_bitmap_entries;
    u32 perf_score = 100;
    /* Modify perf score based on execution time
        ↪  */
    if (q->exec_us * 0.1 > avg_exec_us) {
        perf_score = 10;
    }
    else
    ........
    /* Modify perf score based on bitmap size */
    if (q->bitmap_size * 0.3 > avg_bitmap_size)
        ↪ {
        perf_score *= 3;
    }
    else
    .......
}
```

3.3.2 Distance metric-based score policy. The first improvement is on the basis of distance metric, where we still use these four metrics but greatly decrease the numbers of hyperparameters. We directly use the similarity and distance between the current test case and the average test case to calculate the score. We can define such two vectors, the first vector s_0 is:

$$s_0 = (\text{avg_exec_us}, \text{avg_bitmap_size}, 0, 0) \tag{1}$$

, which represents the average test case and s_q is:

$$s_q = (\lambda_1 * \text{exec_us}, \lambda_2 * \text{bitmap_size}, \lambda_3 * \text{handicap}, \lambda_4 * \text{depth}) \tag{2}$$

, which represents the current test case, where λs are four hyperparameters. Then we use the ratio between these two vectors to represent the distance between the current test case and the average case with such an equation:

$$\text{perf_score} = 100 * \frac{||s_q||_2}{||s_0||_2} \tag{3}$$

We call this method manual tuning as we still need to manually tune $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$.

3.3.3 Reinforcement learning-based score policy. To further help the fuzzer adapt to the score policy, we developed a reinforcement learning-based score policy. During each run of a test case, the fuzzer will use a reinforcement module to automatically tune the λ based on current run and find the best score policy automatically. As we can see from the overall flow of each run of the testcases, it can be naturally formulated as a markov decision process (MDP). As a result, we first formulate the problem into an MDP.

In each run of the instrument, the state is the s_0 and s_q, and each action is simply having the run of the current test case. Then we define the reward on the basis of whether the new test case invokes new hangs or new crushes. For example, if the new test case invokes new hangs and new crushes at the same time, the reward is set to a. If either one of them is invoked, the reward is set to b, otherwise it is set to c.

Then, the core part of our method is how to design the policy network, we set the original hyperparameter λ as the parameter of the policy network $\pi_\lambda(s_t = (s_0, s_q), a_t = a)$. And then we can use the policy gradient to update the λ on basis of reward during with each run of the test case.

$$\begin{aligned}
\nabla_\lambda J(\lambda) &= \mathbb{E}_\lambda[s_t = (s_0, s_q)|a_t = a] \\
&= \mathbb{E}_\lambda[R * \nabla_\lambda \text{perf_score}] \\
&= R * \frac{100}{||s_0||_2} \nabla_\lambda ||s_q||_2 \\
&= \frac{100R}{||s_0||_2 ||s_q||_2} \cdot s_q
\end{aligned} \tag{4}$$

We can calculate the analytic solution of the gradient and during the actual implementation, we directly embed such policy gradient and the calculation of the gradient in the C language in the AFL++ and compile it. In the actual implementation of the fuzzer we use the unsigned 32 to calculate them which is easy to implement and we update the policy network with SGD optimizer where learning rate is set to 0.1 by default. The overall algorithm of our method is shown in Algorithm 1, the loop is straightforward and in each round of fuzzing, we first get the four metrics and update the average vector s_0. Then we update the policy network and calculate out the adapted performance score for each test case, so as to better evaluate the seed.

In our method, we only use the method of policy gradient to test the effectiveness of reinforcement learning-based score policies. While actually, the update of the reward can be very sparse. Because hangs and crashes actually happen occasionally, the occurence of reward c is much more than the occurence of a and b. As a result, policy gradients cannot be the best reinforcement learning method. However, here we want to show how a automatical method can help improve our system and in the latter part, we can see that even if we only use the policy gradient method, it can still bring much benefits. We also encourage readers to try other methods such as PPO [21], Q-learning [27] or propose other novel method to resolve the problem of spare rewards and try other optimizers like Adam [12].

Algorithm 1 The fuzzer with reinforcement learning-based score policy

Input: The fuzzer.
Output: perf_score.
Initialize $(\lambda_1, \lambda_2, \lambda_3, \lambda_4)$, learning rate η, avg_exec_us← 0,avg_bitmap_size← 0
for each run of new test case **do**
  Fuzzing and get q->exec_us, q->bitmap_size, q->handicap, q->depth
  Update avg_exec_us,avg_bitmap_size
  $s_0 \leftarrow (\text{avg_exec_us}, \text{avg_bitmap_size})$
  $s_q \leftarrow (\lambda_1 * \text{q}- > \text{exec_us}, \lambda_2 * \text{q}- > \text{bitmap_size}, \lambda_3 * \text{q}- > \text{handicap}, \lambda_4 * \text{q}- > \text{depth})$
  Set reward $R \leftarrow c$
  if afl->unique_crashes and afl->unique_hangs update **then**
    $R \leftarrow a$
  end if
  if afl->unique_crashes or afl->unique_hangsupdates **then**
    $R \leftarrow b$
  end if
end for
$\lambda \leftarrow \lambda + \eta \nabla_\lambda J(\lambda)$ // Update policy network.
perf_score $\leftarrow 100 * \frac{||s_q||_2}{||s_0||_2}$
Return: perf_score

4 EVALUAION

To evaluate the improvement by multi-level coverage tree structure and reinforcement learning algorithm, we have ported our RLSGD models and manual tuning versions to run on MacBook Pro with Apple M1 Pro chip and 32GB memory. Due to that we did not have much computation resource such as powerful server, we could not contiguously test our work on large scale benchmarks which may take several days to get the results. To address this issue, we developed a light weight benchmark which can quickly get result in a short time, and we also fixed the run time for each experiment at 15 minutes. At last, we used our own benchmark to evaluate some critical performance metrics of each experiment group.

4.1 Benchmark

To conduct an evaluation with limited computation resources, we have the following requirements for our benchmark:

- It should be lightweight. This is the most critical consideration, because we need to get result quickly with limited computation resource.
- It should generates many branches during runtime. To evaluate our work on improving fuzzing coverage, we need many branches to estimate the coverage.
- It should has some hard to find bugs. We have to put some intentional bugs at some branches which won't be easy to reach, and such difficulty will distinguish different experiment groups.

Considering all of the requirements, we decided to implement a simple JSON parser as a benchmark. Without any optimization, a simple JSON parser is pretty easy to implement and it is also lightweight, most test cases can be accomplished within a few seconds.

Since the JSON parser uses tokens to parse a JSON string, it uses a state machine to represent the state of each token. A specific token will lead parsing to a specific state. For example, if it reads a curly open("{"), it recognizes it as a start of an object, then it will reach the object state, and starts to parse it. Thus each token is actually a branch, then there will be many branches generated during run time.

There are also many strict validations in the parsing, for example, number format, boolean value format and so on. So we can put our buggy code under these validations and make it difficult to trigger.

4.1.1 *Benchmark Structure.*

There are four source code files:

- `JSONNode.h`: Some type definitions. No bugs.
- `JSONParser.h`: The main logic of parsing, e.g., parsing object, parsing list and so on. Two crash bugs in the code.
- `Tokenizer.h`: The utility function to read token from input file. One hang bug in the code.
- `main.cpp`: The entry point of the program, read input file and call the parser. No bugs.

4.1.2 *Buggy Code.*

There are three bugs in our benchmark, two of them are crash bugs and one is a hang bug. The crash bugs are in `JSONParser.h` and the hang bug is in `Tokenizer.h`. The buggy code is shown below:

Listing 2: Crash bug in ParseString().

```
std::shared_ptr<JSON::JSONNode> ParseString() {
  printf("Parsing␣String\n");
  std::shared_ptr<JSON::JSONNode> node = std::
      ↪ make_shared<JSON::JSONNode>();
  Token token = tokenizer_.GetToken();
  std::string *sValue = new std::string(token.
      ↪ value);
  /****** BEGIN vulnerable code ******/
  // if string is empty, the node will be
      ↪ deleted, but it mistakely references
      ↪ the
  // nullptr
```

```
  // CRASH
  if (sValue->empty()) {
    delete sValue;
    sValue = NULL;
    node = NULL;
  }
  node->setString(sValue);
  /****** END vulnerable code ******/
  return node;
}
```

Listing 3: Crash bug in ParseNumber().

```
std::shared_ptr<JSON::JSONNode> ParseNumber() {
  printf("Parsing␣Number\n");
  std::shared_ptr<JSON::JSONNode> node = std::
      ↪ make_shared<JSON::JSONNode>();
  Token next_token = tokenizer_.GetToken();
  std::string value = next_token.value;
  /****** BEGIN vulnerable code ******/
  // it does not check the float value, so that
      ↪ if this cannot be converted to
  // float(it starts with more than one '-'), it
      ↪  will delete this node, but it
  // mistakenly try to set the value to 0
  // CRASH
  float fValue = 0.0;
  try {
    fValue = std::stof(value);
  } catch (const std::invalid_argument &e) {
    node = NULL;
  }
  node->setNumber(fValue);
  /****** END vulnerable code ******/
  return node;
}
```

Listing 4: Hang bug in GetToken().

```
auto GetToken() {
  ...
  if (c == '"') {
    token.type = TokenType::kString;
    token.value = "";
    file.get(c);
    /****** BEGIN vulnerable code ******/
    // it does not check if reach the end of
        ↪ file, this will cause a dead loop
    // HANG
    while (c != '"') {
      token.value += c;
      file.get(c);
    }
    /****** END vulnerable code ******/
  }
  ...
}
```

For two crash bugs 2 and 3, both two are caused by access to null portioners and memory corruption. Specifically, 2 is triggered when the string value is empty, in other words, when the string is two consecutive double quotation marks(""), it will be triggered. 3 is triggered when the number value is not a valid negative number, for example, when the number value is "−1.0", it will be triggered.

For the hang bug 4, it is caused by a dead loop. Specifically, when the string value is not closed by a double quotation mark, it will cause a dead loop. Because the code does not check if it reaches the end of a file, when it is out of the end of a file, the condition of the while loop will always be true, then it will cause a dead loop.

These buggy code will cause some crashes and hangs and we will let fuzzers of different experiment groups find them.

4.2 Experiment

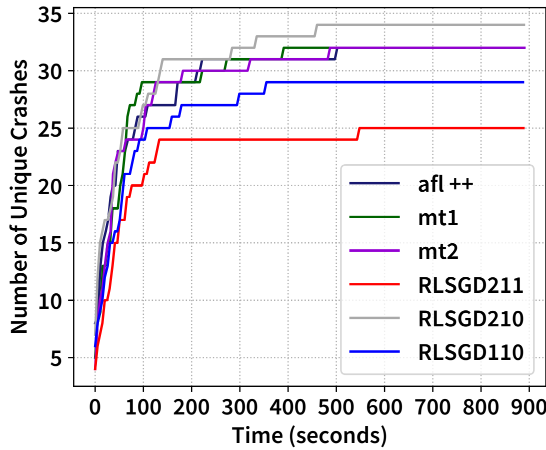
4.2.1 Set Up. We have six experimental groups:

- **AFL++**: The original AFL++.
- **Manual tuning 1**: We manually tuned the parameters of AFL++'s score policy, intentionally make it more aggressive when finding crashes. $(\lambda_1, \lambda_2, \lambda_3, \lambda_4) = (1, 1, 1, 1)$.
- **Manual tuning 2**: Similar to the previous one, but make finding hangs more aggressive. $(\lambda_1, \lambda_2, \lambda_3, \lambda_4) = (2, 1, 1, 2)$.
- **RLSGD211**: Both crash and hang will be rewarded in the model. The parameters of AFL++'s score policy are tuned by the model. $(a, b, c) = (2, 1, 1)$.
- **RLSGD210**: Either crash or hang will be rewarded in the model. $(a, b, c) = (2, 1, 0)$
- **RLSGD110**: Neither crash nor hang will be rewarded in the model. $(a, b, c) = (1, 1, 0)$

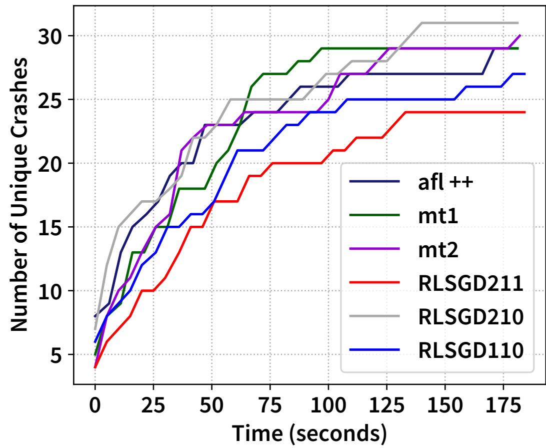
Note that in manual tuning versions, the parameters of AFL++'s score policy are fixed during run time, while in RLSGD versions, the parameters are tuned by the model and can be changed after each fuzzing iteration.

We ran each experiment group to fuzz our own lightweight benchmark for 15 minutes on MacBook Pro with Apple M1 Pro chip and 32GB memory, evaluating the performance in terms of the following metrics:

- **Unique crashes**: The number of unique crashes.
- **Unique hangs**: The number of unique hangs.
- **Total paths**: The number of total paths covered in run time.



(a) 15 minutes.



(b) First 3 minutes.

Figure 4: The number of unique crashes found in 15 minutes.

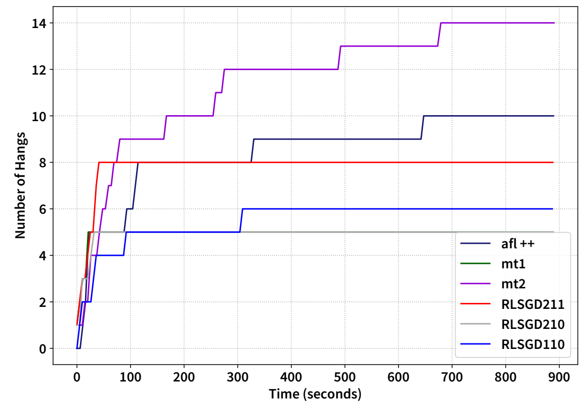


Figure 5: The number of unique hangs found in 15 minutes.

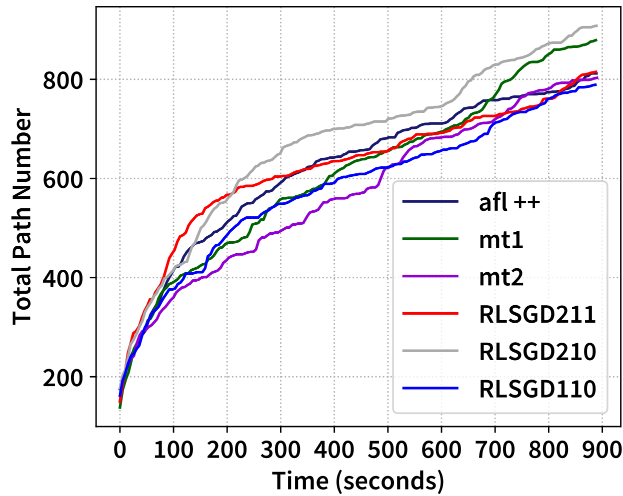


Figure 6: The number of total paths found in 15 minutes.

4.2.2 Results. We collected fuzzing states of run time and generated some graphs for the metrics mentioned above.

For unique crashes: From Figure 4a, we can see that RLSGD210 is the winner of finding unique crashes, since it finds out the most crashes. From Figure 4b, we can see that RLSGD210 grows quickly at beginning, especially in first 25 seconds, it grows fastest. This indicates that RLSGD210 finds most crashes and it can also reach saturation more quickly.

For unique hangs: From Figure 5, we can see that manual tunning version 2 (mt2) finds out the most hangs while the winner of finding crashes RLSGD210 finds out the least hangs. The reason why mt2 wins we attribute to our tunning for finding hangs. The reason why RLSGD210 performs worst we attribute to our model's trade off between finding crashes and hangs may be too aggressive.

Fuzzer	Crash	Hang	Total path
AFL++	32	10	819
Manual tuning 1	32	5	890
Manual tuning 2	32	14	803
RLSGD211	25	8	820
RLSGD210	34	5	951
RLSGD110	29	6	797

Table 1: Results summary.

For total paths: From Figure 6, we can see that RLSGD210 has an overall better coverage than others.

To sum up, the final results of the number of unique crashes and hangs, and the number of total paths are shown in Table 1.

4.3 Experiment Conclusion

The RLSGD210 is the winner of coverage and finding crashes, but it is not good at finding hangs for now, we can further tune the model to get better overall performance. Manual tuning 2 seems to have a more balanced performance, but its parameters are hardcore and fixed during run time, as a result we do not think it has any more optimization space.

Both manual tuning 2 and RLSGD210 have obvious improvement compared to original AFL++. Although manual tuning 2 has 1.95% creep of coverage, it has 40% increase of finding hangs. Although RLSGD210 has 50.00% creep of finding hangs, it has 6.25% increase of finding crashes and 16.12% increase of coverage, it also seeks crashes more quickly than AFL++ in early stage. Thus, multi-level coverage tree structure has been improved by AFL++.

We believe that AFL++'s hardcode score policy lacks flexibility and sometimes may not be intelligent enough to enlarge coverage or dig deeper. Thus, using reinforcement learning algorithms to dynamically adapt score parameters is promising. RLSGD210 has proved that it can improve coverage and find crashes. Next step, we will try to eliminate the defects of finding hangs, try to use ADAM optimizer or other algorithm like PPO to make further optimization.

5 LESSONS LEARNED

Our course project progress is demonstrated in Figure 7. We firstly tried to come up with some ideas from reviewing previous high-quality papers. For instance, one of them is CollAFL [7], which we have carefully reviewed and analyzed during our lectures. On the basis of our paper reading and discussion, we agreed that the structure of multi-level coverage tree [26] can address the code coverage issues effectively and efficiently, so we further read some related papers in order to try to validate its neccecioty and propose a further solution. However, we found that AFL++'s score calculation was hardcode and parameters seemed all magic numbers. Thus we tried to take advantage of state-of-the-art reinforcement learning approaches with multi-level coverage to dynamically adapt these score parameters according to the reward of crash and hang. Finally, due to the limitation of computation resource, we developed our own lightweight benchmark Jesting-SON to evaluate work.

From the midterm evaluation, we figured out the clear path of how to produce our final projects. As a result, after midterm evaluation, we re-examined our proposal and our progress by midterm and made a plan of what to do next step. Then, in the rest of time, we addressed the issues we peoposed and made modifications on the evaluation part due to our limited resources. It is important to do some changes based on the actual requirements and we still can meet our initial expectation of our project.

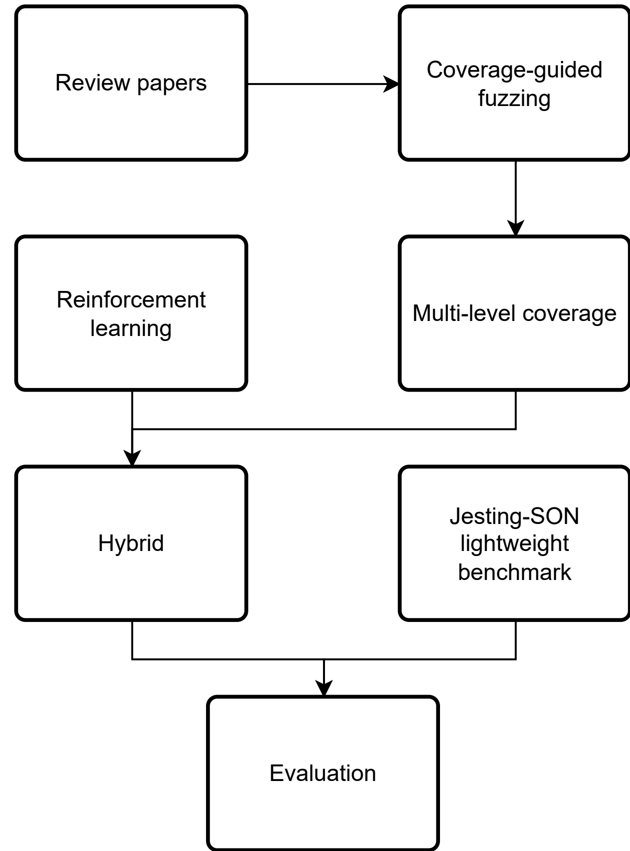


Figure 7: Our course project flow chart.

6 CONCLUSION

In this paper, we proposed a two-fold hybrid solution to tackle the unresolved challenges posed in the area of coverage-guided greybox fuzzing. We leverage a multi-level coverage tree to take advantage of various coverage metrics with efficiency, which significantly makes the new code coverage metrics more informative. Meanwhile, state-of-the-art reinforcement learning algorithms are being leveraged for intelligently scoring the seeds. Hence, we achieved a better trade-off between seed exploration and seed exploitation, which further improve the performance of fuzzers. Evaluations of our work are conducted on a lightweight JSON parser benchmark, which is implemented for experiments under low computation budgets. More importantly, it reveals the superiority of our approach on the basis of unique crashes, unique hangs, and total covered paths.

7 ACKNOWLEDGMENTS

We would like to thank Prof. David Lie, because he delivers perfect lectures in terms of the state-of-art fuzzing technology. By doing course project, we have learned a lot during coding, brain storm, paper reading and any other things we did to complete this course project. As M.A.Sc. Students, this is a good practice experience.

REFERENCES

[1] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[3] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[4] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278.

[6] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. 2014. KameleonFuzz: evolutionary fuzzing for black-box XSS detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. 37–48.

[7] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[8] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.

[9] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.

[10] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*. 45–48.

[11] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2511–2513.

[12] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[13] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 1–13.

[14] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[15] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[16] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

[17] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*. PMLR, 4901–4911.

[18] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. {ParmeSan}: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2289–2306.

[19] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.

[20] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[21] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[22] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[23] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[24] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[25] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[26] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. (2021).

[27] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3 (1992), 279–292.

[28] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 511–522.

[29] Michal Zalewski. 2017. American fuzzy lop.

[30] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. {FuzzGuard}: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *29th USENIX Security Symposium (USENIX Security 20)*. 2255–2269.