Improving Directed Greybox Fuzzing using Monte-Carlo Decision Tree and Ant Colony Optimization

Taeeun Kim, Subin Kim, Geonho Koh, and Sungsoo Han

School of Computing, Korea Advanced Institute of Science and Technology
{goodtaeeun, 21supersoo, ghkoh97, lomotos10}@kaist.ac.kr

Abstract. AFLGo suggests Directed Greybox Fuzzing (DGF) which thinks of reducing the reachability to target location as an optimization problem and tries minimize the distance of the generated inputs to the targets. However, the transition point between exploration phase and the exploitation phase should be given by the user, and calculates fitness based on the distance which can easily fall to a local optima despite using SA. Thus, we introduce AFLGOpt, which uses Monte-Carlo decision tree based on pheromone of each seed, which is a changeable variable that prevents from the fuzzing to fall into local optima. The proposed model proves to be at most eight times faster in finding valid crashes than the original AFLGo, and fewer timeout results in overall procedure.

Keywords: Directed greybox fuzzing · Monte-Carlo decision tree · Ant colony optimization

1 Introduction

Fuzzing is a software testing technique that automates testing a computer program with unexpected or random data in purpose of finding the vulnerability. Greybox fuzzing (GF) is thought as the state-of-the-art techniques in vulnerability detection. It instruments through the program with a given seed input, and new inputs that is generated by mutating previous inputs when they provide new interesting paths are added to the queue of the fuzzer. American Fuzzy Lop (AFL)[2] is considered as one of the best GF techniques, and discovered numerous significant vulnerabilities. However, fuzzing through undirected random inputs can cause significant overhead, which results in delayed time of testing. A directed fuzzer, unlike undirected fuzzers, spend the time mostly on finding specific target locations without wasting resources by running through unnecessary program components.

As a result, AFLGo[1] suggested Directed Greybox Fuzzing (DGF) which thinks of reducing the reachability to target location as an optimization problem and tries minimize the distance of the generated inputs to the targets. To compute seed distance, AFLGo computes and the distance of each basic block to the targets at compile-time. At runtime, AFLGo minimizes seed distance using

Simulated Annealing(SA) through power schedule, which is called the exploration phase. A power schedule controls the energy of the seeds, and the energy determines the time spent fuzzing the seed, which is the exploitation phase. Similar with all greybox fuzzing techniques, by aborting time-consuming analysis to compile time, overhead at runtime is reduced.

However, the balance between exploration phase and the exploitation phase is determined as input that should be given by the user, and it is not automated. Also, ALFGo calculates fitness based on the distance to the target location, which is a fixed location, and can easily fall to a local optima despite using SA.

Thus, we introduce AFLGOpt, which uses Monte-Carlo decision tree based on pheromone of each seed, which is a changeable variable. We provide a new perspective of the seed, where from the flat structured fuzzer's queue of seeds, we draw a Monte-Carlo tree structure. Each node represents a seed, and the edge represents a parent-child relationship. We select a node from the seed queue and generate mutant seeds based on the pheromone value of the seed. Based on this graph representation, if the mutant discovers new coverage or proves to be interesting, we update the pheromone level of the mutant as well as the pheromone of its ancestors.

The strength of AFLGo is that it employs the knowledge of the distance information. However, in the exploration phase, the knowledge is not used and initially treats all seeds equally in order to avoid the local optima. However, AFLGOpt utilizes the distance information from the start, and seeds are treated differently based on the pheromone value, which is affected by the distance and coverage. Fitness score of the seed is evaluated with distance score and the pheromone level, which is not fixed, and more prone to avoiding local optima. When there is a useless seed with high distance score, the pheromone will decrease, resulting in lower fitness score, and increasing the possibility to search more various seeds. In addition, the AFLGOpt does not rely on explicit user inputs for phase transition.

Our contributions are as the following:

- The integration of directed greybox fuzzing and monte-carlo decision tree
- Reducing the possibility of falling into the local optima by employing the concept of pheromone
- The implementation of AFLGOpt which is publicily available at https://github.com/goodtaeeun/FuzzFrame

2 Related Work

Undirected/Directed Greybox Fuzzing Undirected greybox fuzzing fuzzing is one of the original state-of-the-arts in vulnerability detection. The method determines a unique identifier for the path that is exercised by an input, with almost negligible performance overhead. New inputs are generated by mutating a provided seed input and added to the fuzzer's queue if they exercise a new and interesting path. [1] The leading implementation of undirected greybox fuzzing is AFL[2]. AFL maintains a global map of tuples seen in previous executions.

When a mutated input produces an execution trace containing new tuples, the corresponding input file is preserved and routed for additional processing later on. Inputs that do not trigger new local-scale state transitions in the execution trace (i.e., produce no new tuples) are discarded, even if their overall control flow sequence is unique. This approach allows for a very fine-grained and long-term exploration of program state while not having to perform any computationally intensive and fragile global comparisons of complex execution traces, and while avoiding the scourge of path explosion.

Existing studies have added directed fuzzing to existing greybox fuzzers. Applications of directed fuzzers include patch testing, crash reproduction, static analysis report verification, and information flow detection. The current state-of-the art implementation of directed greybox fuzzing is AFLGo [1], which adds directedness to the existing greybox fuzzer implementation of AFL. AFLGo first calculates the function level target distance using the function call graph. AFLGo identifies the target functions in the call graph, and for each function, computes the harmonic mean of the length in the shortest path to the target. Then, AFLGo extracts the basic block level distance using the control flow graph. It first finds the target basic block and assign its distance as 0. It then finds all the basic blocks that calls functions, and assign their weights as some constant. The constant can be tuned and does not need to be exact. Afterwards, it can compute harmonic mean of the sum of the (length of the shortest path to any function calling basic block) and the (tuned constant of the basic block).

The above procedures are pre-run on compile-time. At runtime, AFLGo focuses on finding the seed distance from the instrumented binary. It finds the basic blocks that the seed traversed, and computes the seed distance, which is calculated by dividing (aggregated pre-computed basic block level distance values) with the (number of executed basic blocks).

Since greybox fuzzing is a mutation-based fuzzing, it is divided into two main steps: selection and mutation. In selection, seeds that occur crashes, or which is interesting will be selected. The term *interesting* denotes that a seed has found a new path. Before mutation, AFLGo assigns *energy* to the seed, which is the main idea of this AFLGo. *Energy* represents the number of new seeds that are mutated from the seed. To calculate the energy that will be assigned, AFLGo adopts simulated annealing in order to avoid falling into local minima. As it is well known, simulated annealing will sometimes assign more energy to further away seeds.

Ant Colony Optimization Ant colony optimization is a bio-inspired algorithm that aims to balance exploration and exploitation without need of tuning of hyperparameters from outside sources. It is mainly used for graph-related problems (e.g. TSP). The simulated ants in the algorithm leave a pheromone trail on explored edges; since there is no guarantee that the initial group ants find the best path, the pheromone is set to evaporate by a set amount as time passes.

We explain the general algorithm of ant colony optimization that is targeted towards TSP. To initialize the graph, drop ants on random nodes on the graph. Also, deposit a small amount of pheromone on all edges uniformly. Ants choose which edge to cross probabilistically, considering the length of the edge and the amount of pheromone on the edge. When ants finish a tour, they retrace their tour, depositing pheromone in an amount inversely proportional to the length of the route. Before starting the next cycle of trips, the pheromone evaporates by a set amount. Eventually, the ants are shown to converge on the shortest path.

Monte Carlo Decision Tree Monte Carlo Tree Search is a heuristic search algorithm that is mainly used for traversal of game trees. In particular, it is mainly used in combinatorially explosive board games such as Go[4], chess, and shogi[5], and games with incomplete information such as poker.[6] Monte Carlo Tree search focuses on expanding the search tree based on random sampling of the search space.

The algorithm of Monte Carlo Tree search is as follows. Each round consists of the following four steps: Selection, Expansion, Simulation, and Backpropagation. Selection starts from the root of the search tree, and selects successive child nodes until a leaf node is reached. The method of actually selecting the child nodes differs on the specific version of Monte Carlo Tree Search. Expansion then creates one or more child nodes from the leaf nodes, provided that the leaf node does not immediately end the game. Simulation completes one random playout from the node chosen in Expansion. Backpropagation uses the result of the Simulation playout to update the information in the nodes in the path from the child node to the root node.[7]

While the Monte Carlo method is not directly related to greybox fuzzers or directed fuzzers, it is included in this section to provide a background our own developed method, which bears some similarities to Monte Carlo Tree Search.

3 Implementation

The main idea of AFLGOpt is replacing simulated annealing part to another model without external adjustment of hyperparameter. Since SA(Simulated Annealing) algorithm was applied to avoid falling into local minimum, new model also focused on this part.

3.1 ACO Inspired Greybox Fuzzing

We adopted the concept of 'pheromone' from ACO(Ant Colony Optimization). However, since our problem description is not suitable for vanilla ACO, some modifications have been made. First, the concept corresponding to 'ants' and 'path' was decided. Mutants generated in each iteration can be thought of as ants that give feedback(pheromone) to their parent seeds when they find an interesting path(seed). And seeds with pheromone information can be considered as a path to the final target. In order to find the input directed to the final

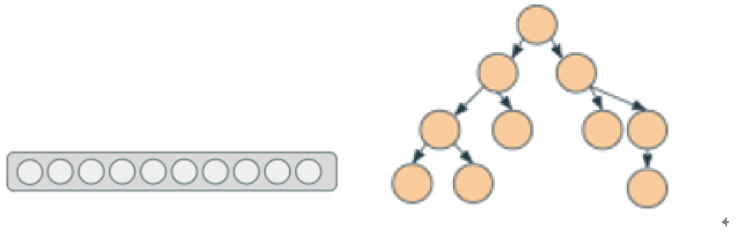
(a) seeds structure in AFLGo (b) seeds structure in AFLGOpt

Fig. 1: Modified seeds structure

target, a chain of mutations occurred from the initial seed, and seeds in the chain contributed to finding the target to some extent. Therefore, if a mutant finds an interesting seed, it reinforces all the pheromones of its ancestor seeds, which is why each seed functions as a path.

In addition, vanilla ACO continues to search until ant finds food, however, ant of AFLGOpt has a fixed limit to search. Once a mutant seed is created, the point at which it can be reached is determined, so it cannot proceed further. But since we know the distance to the target, information about how far the ant has reached can be delivered to the path(seeds) it has gone through. This can be compared to the fact that each ant has a limit to the distance it can travel due to its low physical strength, but the smell of target food is strong, so it can convey how close it has reached.

Several pieces of information have been added to enable this implementation. First, pheromone, a value indicating how likely each seed is to generate an interesting mutant, has been added to each seed. Second, a pointer indicating which seed it has mutated from, that is a pointer of parent seed, was added. This results the set of seeds, which was in the form of a linear queue, appear in a tree-shaped structure as shown in the figure.

3.2 Pseudo-code implementation

The following algorithm shows the AFLGOpt algorithm with a slight modification of the Greybox fuzzing algorithm. ASSIGNENERGY in line 3 of Algorithm1 was modified, and UPDATEPHEROMONE in line 12 of Algorithm1 was added.

Assign Energy Energy p is a variable that determines how many mutants will be generated in this iteration, and the higher the probability that the mutation will approach the target, the higher the value. That is, basically the closer the distance to the target, the higher the energy. However, there is a high possibility of being trapped in a local minima, so there is a need for a way to avoid it. In

Algorithm 1 AFLGOpt

```
1: repeat
2:     $s$ = CHOOSENEXT($S$)
3:     $p$ = ASSIGNENERGY($s$)
4:     for $i$ from 1 to $p$ do
5:         $s'$ = MUTATEINPUT($s$)
6:         if $t'$ crashes then
7:             add $s'$ to $S_x$
8:         else if ISINTERESTING($s'$) then
9:             add $s$ to $S$
10:        end if
11:    end for
12:    UPDATEPHEROMONE($s$)
13: until timeout reached or abort-signal
```

Greybox Fuzzing, it is Simulated Annealing(Algorithm 2), and in AFLGOpt, it is pheromone(Algorithm 3).

Algorithm 2 ASSIGNENERGY in Directed Greybox Fuzzing

```
1: $T$ = TEMPERATURE($currentTime$)
2: $powerfactor$ = SIMULATEDANNEALING($T$, $s.distance$)
3: $energy$ = 100 * $powerfactor$
4: return $energy$
```

Algorithm 3 ASSIGNENERGY in AFLGOpt

```
1: $perf\_score = s.pheromone/s.distance$
2: $energy$ = 100 * $perf\_score$
3: return $energy$
```

Update Pheromone The value of Pheromone varies depending on how many interesting mutants were generated from the seed. Since not only themselves but also parents contributed to the creation of interesting mutants, pheromones increase in part, but not as much as the seed that generated the mutation themselves. In addition, if no more interesting mutations occur in some branches, pheromone evaporates so that their importance can be reduced.

3.3 Monte Carlo Tree Search

AFLGOpt was approached with the concept of pheromone at first, however, it was confirmed that the final output of the model resembles MCTS rather

Algorithm 4 UPDATEPHEROMONE

```
$s.pheromone$ *= $EVAPORATION\_RATE$
$importance = s.numInteresting/avgNumInteresting$
$node = s$
**while** $node.parent$ is not NULL **do**
    $importance$ *= $importance$ $CONTRIBUTION\_RATE$
    **if** $importance$ ¿ 1 **then** $node.pheromone$ *= $importance$
    **end if**
**end while**
**if** $s.pheromone$   1 **then** $s.pheromone = 1$
**end if**
```

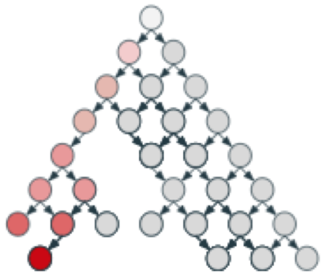

Fig. 2: finding interesting seed will also increase parent seeds' pheromone

than ACO. The 'selection' step of the MCTS may correspond to the 'Assign Energy' step of the AFLGOpt. One difference is that AFLGOpt, unlike MCTS can be mutated even in branch nodes. Also, the 'expansion' and 'simulation' step of the MCTS may correspond to the 'mutation' and 'Calculate Score' step, respectively. Finally, the 'backpropagation' step of MCTS is almost consistent with the 'Update pheromone' step of AFLGOpt, except that the contribution decreases toward the parents. As a result, AFLGOpt got the idea from ACO's pheromone, but ultimately a model closer to MCTS was designed.

4 Evaluation

To evaluate the soundness of our approach, we conducted a comparison experiment. We implemented our approach directly on top of the existing directed greybox fuzzer, AFLGo and named our tool AFLGOpt. We tested these two tools, AFLGo and AFLGOpt, on the perspective of crash reproduction. All the implementation and the experiment settings are in our github repository[3].

4.1 Experiment Settings

To compare AFLGOpt to the baseline AFLGo, we choose the vulnerabilities in Binutils that were used to evaluate AFLGo in its original paper. These vulnerabilities are identified by the CVE-IDs. We set a timeout of 6 hours . For AFLGo, the exploration time was set to 6*7/8 hours, to be coherent with the ratio of time budget and the exploration time as reported in AFLGo's paper, which was 8 and 7 hours respectively. For AFLGOpt, 7 versions with different parameter values were used to evaluate the impact of each parameter. We repeated the experiment for 8 times. Experiment was run on a cpu server of 64 cores. Maximum of 48 cores were used in parallel, each bound with a single docker instance.

We use the Time-to-Exposure to evaluate the fuzzing performance. Time-to-Exposure (TTE) measures the time taken to generate a first input that triggers the targeted bug. In order to determine which input triggers which bug, we instrumented the target binary with ASAN options. If an ASAN log of two crashing inputs are same, or delivers identical meaning, they are considered to expose the same bug. Thus, if an input is related to an already existing POC of a known bug, it is considered to expose the same bug. And the time taken to generate the first of such inputs is recorded as the TTE. Since fuzzing has innate randomness in its execution, we took the median, rather than the mean, value of TTE out of 8 repetition results

From now on we will address the research questions that we wanted to answer through this project.

4.2 RQ1

RQ1. How faster is AFLGOpt compared to its baseline, AFLGo?

Table 1: Comparison of TTE for each bug with AFLGOpt and AFLGo

bug	AFLGo	AFLGOpt							largest difference
		A: initial pheromone(1.0) / B: parent decrease(0.8) / C: evaporate rate(0.9) / D: minimal pheromone(0.1)							
		no change	A to 0.8	B to 0.9	B to 0.95	C to 0.8	C to 0.95	D to 0.2	
2016-4487	2535	583	476	617	707	316	602	493	391
2016-4489	6002	1043	877	1446	948	1562	1545	1782	905
2016-4490	1117	141	172	291	277	197	257	209	150
2016-4491	14905	4113	3625	4241	3489	3667	3669	3705	752
2016-4492	4834	3582	3620	4063	2707	5593	2226	2921	3367
2016-6131	T/O	7536	13402	10090	9695	9482	7070	7467	6332

As we can see in Table 1, all version of AFLGOpt generally performs better than AFLGo. The best performance boost happens in the CVE-2016-4490, by the original version of AFLGOpt, which is almost 8 times faster than AFLGo. There is only one case where AFLGo outperforms AFLGOpt, CVE-2016-4492. However, in this case, AFLGo resulted in two timeouts out of 8 repetitions, while none of the versions of AFLGOpt resulted in timeout in this case. If we are to choose a single version that performs good for all of the cases, it would be the one where evaporation rate is set to 0.95, This version of AFLGOpt exposed the targeted bugs four times faster than the AFLGo. We can also see that AFLGOpt is more free from the threat of local minima, since it has much fewer timeout results compared to AFLGo.

4.3 RQ2

RQ2. How sensitive is the performance of AFLGOpt to the parameter values?

In order to investigate the impacts of each parameter on the performance, we made and tested the variants of AFLGOpt. As we can see in the results, modifying any of the parameters do impact the performance, some positively, some negatively. Thus, all parameter is relevant to the performance.

If the performance is unstable due to the parameters, it is a threat to our validity, because we cannot ensure that it will work well on other target programs. However, only one version of AFLGOpt performed inferior to AFLGo in only one bug case. In other cases, the worst performance of AFLGOpt was still at least 3 times faster than the AFLGo. The implies that performance of AFLGOpt does depend on parameter values, but any reasonable value would result in better performance than AFLGo.

4.4 RQ3

RQ3. How many mutants are added to the seed?

In order to evaluate the directedness, we can measure the ration of entire mutants versus the muatnats that actually discovered new coverage, in other

Table 2: Meaningful mutants ratio of AFLGo and AFLGOpt

bug	AFLGo	AFLGOpt						
		A: initial pheromone(1.0) / B: parent decrease(0.8) / C: evaporate rate(0.9) / D: minimal pheromone(0.1)						
		no change	A to 0.8	B to 0.9	B to 0.95	C to 0.8	C to 0.95	D to 0.2
2016-4487	0.0045	0.0086	0.0056	0.0053	0.0052	0.0052	0.0052	0.0055
2016-4489	0.0042	0.0050	0.0050	0.0052	0.0053	0.0048	0.0051	0.0070
2016-4490	0.0029	0.0033	0.0034	0.0035	0.0034	0.0067	0.0035	0.0034
2016-4491	0.0031	0.0034	0.0033	0.0035	0.0034	0.0035	0.0035	0.0035
2016-4492	0.0043	0.0054	0.0051	0.0053	0.0052	0.0050	0.0056	0.0050
2016-6131	0.0048	0.0053	0.0051	0.0050	0.0050	0.0051	0.0052	0.0055

words, added to the queue. The ratio is given in the Table 2. As expected, all versions of AFLGOpt generates more mutants that are meaningful, which means that it is more directed than AFLGo.

5 Conclusion

In conclusion, We implemented AFLGOpt with the approach of improving the search algorithm with pheromone concepts. AFLGOpt performs better than AFLGo by four times. It is more free of local minima and is more directed than AFLGo. Our evaluation may not be a solid evidence because we did not test our approach on other programs. However, since we tested on different bug targets, and ran multiple iterations, we have an evidence that this result may be generalized.

References

1. Böhme, Marcel, et al. "Directed greybox fuzzing." Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017.
2. https://github.com/google/AFL
3. https://github.com/goodtaeeun/FuzzFrame
4. Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature*. **529**, 484-489 (2016,1)
5. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. & Others Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv Preprint ArXiv:1712.01815*. (2017)
6. Rubin, J. & Watson, I. Computer poker: A review. *Artificial Intelligence*. **175**, 958-987 (2011)
7. Chaslot, G., Winands, M., HERIK, H., Uiterwijk, J. & Bouzy, B. Progressive strategies for Monte-Carlo tree search. *New Mathematics And Natural Computation*. **4**, 343-357 (2008)