SLOPT: Bandit Optimization Framework for Mutation-Based Fuzzing

Yuki Koike [1], Hiroyuki Katsura [1], Hiromu Yakura [2], Yuma Kurogome [1]

[1] {yukik, hiroyukik, yumak}@ricsec.co.jp
[2] hiromu.yakura@aist.go.jp

[1] Ricerca Security, Inc., Tokyo, Japan
[2] University of Tsukuba, Ibaraki, Japan
[2] National Institute of Advanced Industrial Science and Technology (AIST), Japan

Fuzzing has become a widespread method to find bugs

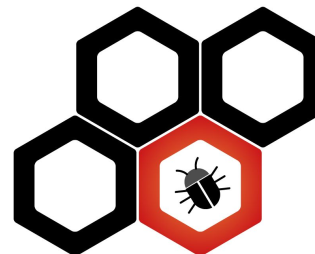
Thanks to its automaticity, fuzzers are

- easy-to-use in CI/CD.
- highly scalable.

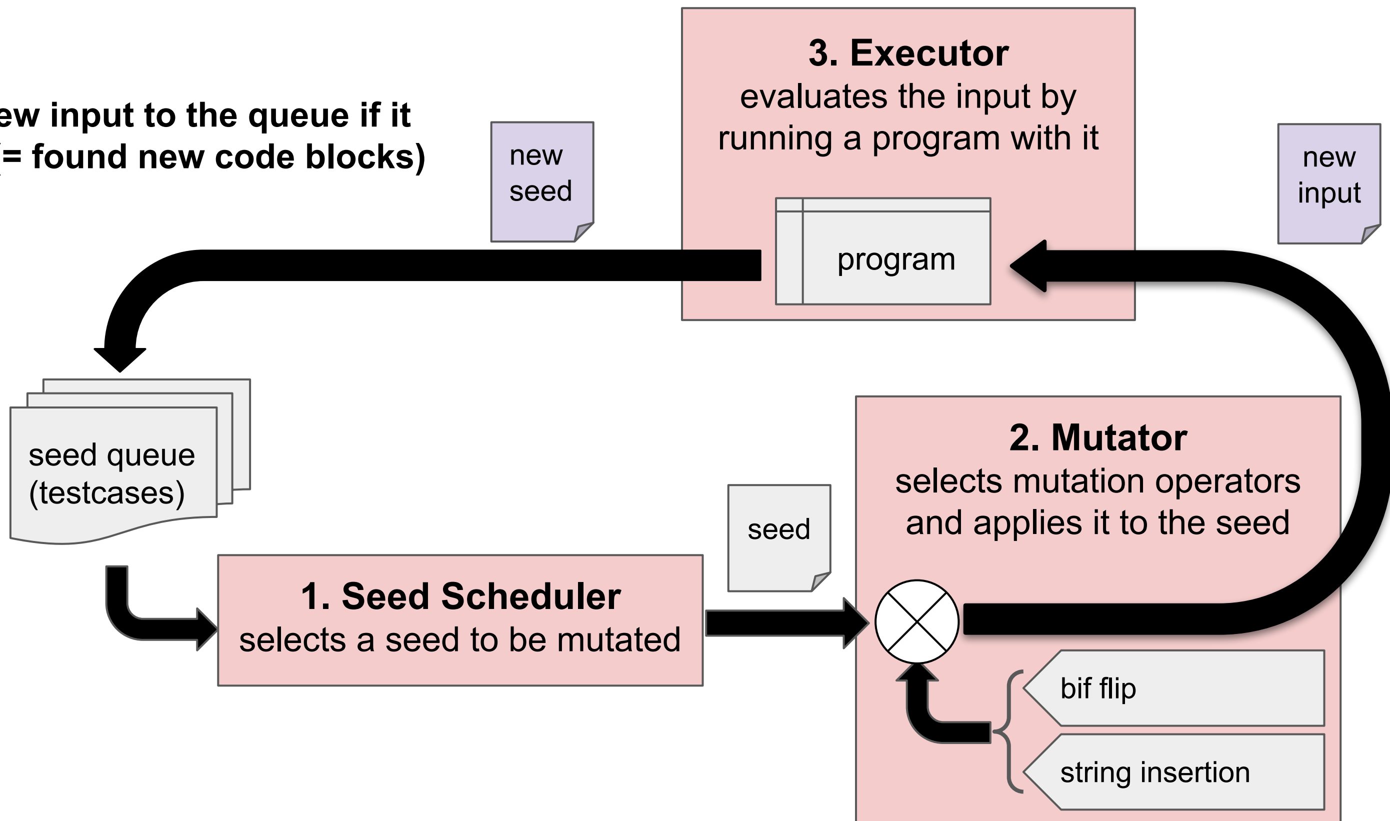
These characteristics made fuzzing in popular use.

As of July 2022, OSS-Fuzz has found over 40,500 bugs in 650 open source projects.

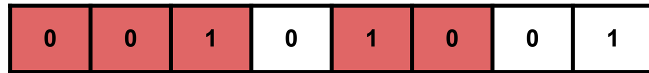
As of May 2022, ClusterFuzz has found 25,000+ bugs in Google (e.g. Chrome).



Mutation-based fuzzing

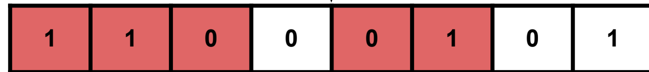


Mutation operators

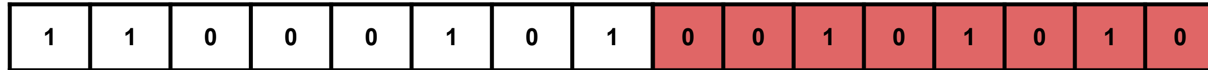


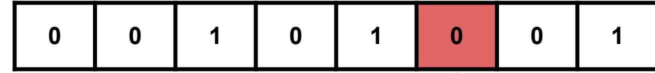
Arith Little End

+15

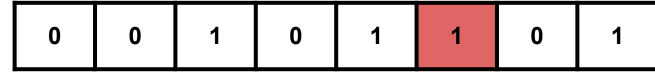


Insert/Delete





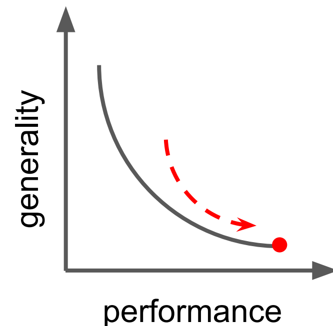
BitFlip



Performance tuning (specialization) vs. generality

Fuzzer performance improves when tuned for specific programs.
E.g.

- Structure-aware fuzzing
  - considers file format during mutation not to create unfruitful invalid inputs.
- Human-in-the-loop fuzzing
  - Human analyst adds special annotations by hand to boost fuzzing.



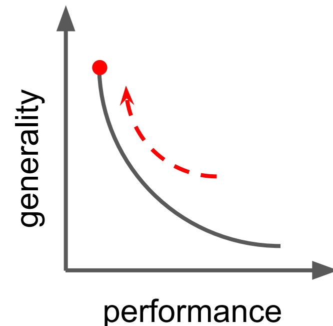
But such performance tunings can spoil generality and scalability.

- Employing them may narrow the range of applicable target softwares.
- It is unrealistic for us to take time to tune fuzzers for each of thousands of open source projects.

Performance tuning (specialization) vs. generality

Fuzzer performance improves when tuned for specific programs.
E.g.

- Structure-aware fuzzing
  - considers file format during mutation not to create unfruitful invalid inputs.
- Human-in-the-loop fuzzing
  - Human analyst adds special annotations by hand to boost fuzzing.



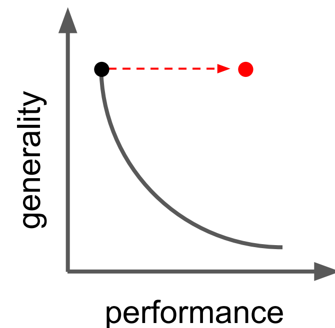
But such performance tunings can spoil generality and scalability.

- Employing them may narrow the range of applicable target softwares.
- It is unrealistic for us to take time to tune fuzzers for each of thousands of open source projects.

Related work: online optimization for mutation

There are studies employed online optimization in mutation to boost fuzzing performance while keeping the generality.

- Some of them found that effective mutation operators are different depending on programs.
  - This fact serves as the foundation of these studies.
- Most related existing studies:
  - with bandit: Karamcheti et al., CMFuzz, HavocMAB
  - with heuristics: MOPT



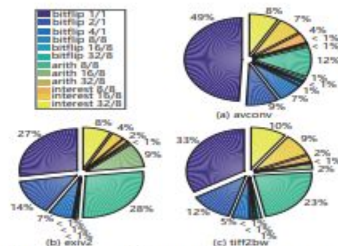


Figure 3: Percentages of interesting test cases produced by different operators in the deterministic stage of AFL.

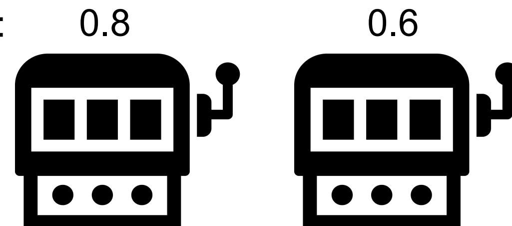
From "MOPT: Optimized Mutation Scheduling for Fuzzers." Chenyang et al. 2019. In Proceedings of the 28th USENIX Security Symposium (Security'19).

Bandit problem

Basic definition of bandit problem:

- There is the predetermined set of choices.
- Once picked, each choice returns some rewards with an unknown probability.
- A player needs to pick one choice in each round and maximize the total rewards earned over T rounds.

Expected rewards:



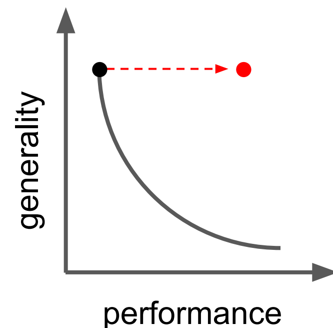
Bandit algorithms efficiently maximize rewards.

In fuzzing, choice is mutation operator and reward is whether input is valuable.

Our interest: making effective use of bandit algorithms

Our motivation is also improving fuzzer performance with generality.
We use bandit algorithms to this end because optimizing mutation can be naturally interpreted as bandit problem.

To fully draw the power of bandit algorithms, we made **three observations** before incorporating bandit algorithms into fuzzers.



Observations#1: mutation scheme

While the previous studies observed that mutation operators have different efficiency, they apply multiple different operators to a single input in mutation.

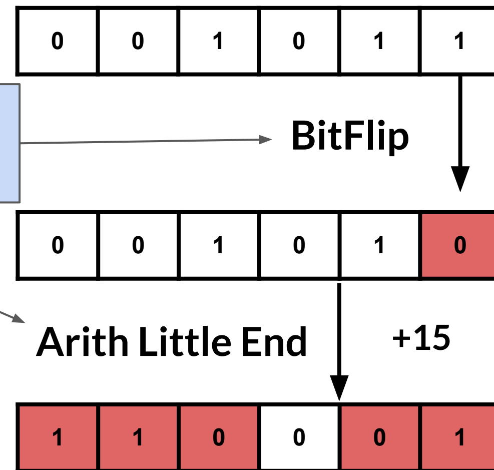
This makes it difficult to analyze which operator was actually effective later.

Algorithm 1 Conventional Random Mutation Scheme

Input: *seed* – a test case to be mutated
Output: *input* – a new input to be tested

```
function RandomMutation(seed)
    input ← CopyBytesFromSeed(seed)
    batch_size ← DecideBatchSize()
    for i ← 1 to batch_size do
        mutation ← SelectOperator()
        pos ← SelectPosition(input)
        input ← ApplyOperator(mutation, input, pos)
    end for
    return input
end function
```

The new input was valuable.
But which was contributed?



Observations#1: mutation scheme

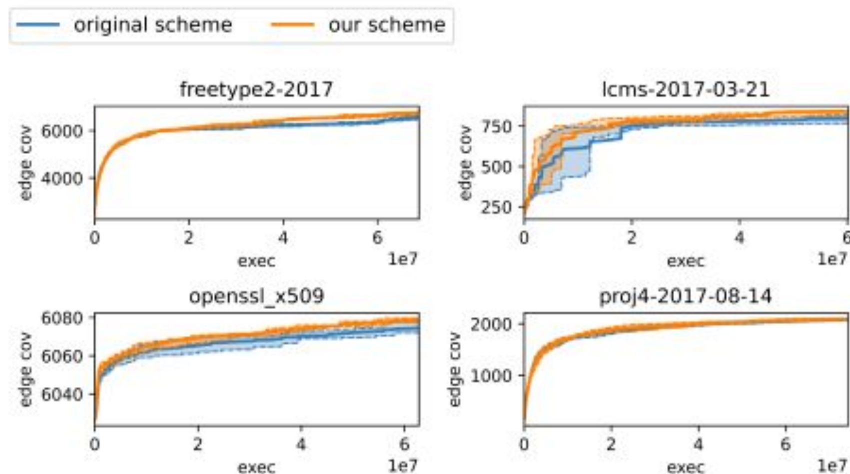
We modified the mutation scheme so that only one type of mutation operator is applied to a single input.

Empirical experiments revealed this modification does not seriously affect the performance. Thus, there is no need to mix different mutation operators.

Algorithm 2 Our Random Mutation Scheme

Input: *seed* – a test case to be mutated
Output: *input* – a new input to be tested

```
function RandomMutation(seed)
    input ← CopyBytesFromSeed(seed)
    batch_size ← DecideBatchSize()
    mutation ← SelectOperator()
    for i ← 1 to batch_size do
        pos ← SelectPosition(input)
        input ← ApplyOperator(mutation, input, pos)
    end for
    return input
end function
```



Observations#2: batch size

Batch size is a parameter meaning how many operators applied to one input.
We ran AFL++ to see which batch size tended to produce valuable inputs. We found

- distributions of effective batch size are different **depending on programs**.
- It also **depends on seed size**.
  - This means optimization should be done independently for different seed sizes.

Thus, batch size also should be an optimization target.

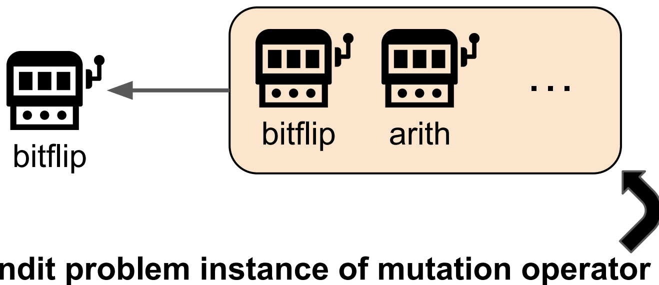
Algorithm 1 Conventional Random Mutation Scheme

Input: *seed* – a test case to be mutated
Output: *input* – a new input to be tested

```
function RandomMutation(seed)
    input ← CopyBytesFromSeed(seed)
    batch_size ← DecideBatchSize()
    for i ← 1 to batch_size do
        mutation ← SelectOperator()
        pos ← SelectPosition(input)
        input ← ApplyOperator(mutation, input, pos)
    end for
    return input
end function
```

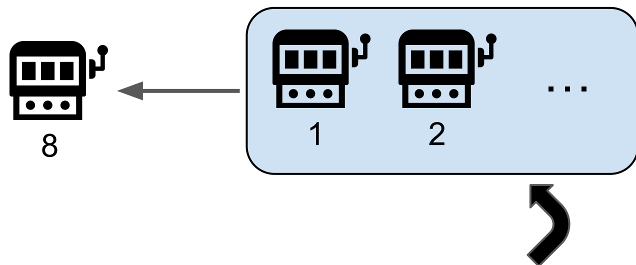
Our proposed algorithm: SLOPT

1. Select mutation operator.



bandit problem instance of mutation operator

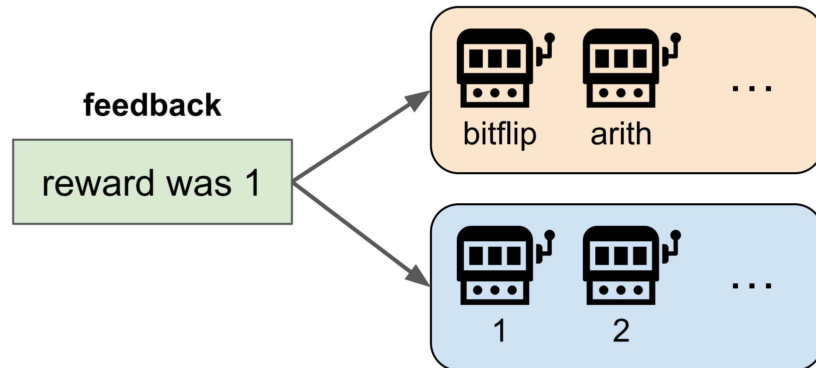
2. Select batch size.



instance of batch size used if operator == bitflip and len(seed) ∈ [100, 1000)

3. Mutate seed with selected parameters.

4. Check if new input was valuable and update problem instances.



Observations#3: optimization algorithm difference

The aforementioned existing methods use different optimization algorithms such as KL-UCB and Thompson Sampling.
Whether adopting other algorithms affects the performance was not mentioned.

We had a preliminary experiment of optimizing AFL++ with different algorithms.
It showed the difference of algorithms somewhat affects the performance.

Table 2: Ranks and scores averaged over 10 PUTs of AFL++ and 8 versions of SLOPT-AFL++ after 24 h.

	AFL++	UCB1	KL-UCB	TS	dTS
Rank Avg	*7.9*	5.5	3.5	3.9	3.0
Score Avg	*89.44*	93.83	96.58	**98.81**	97.85
	dBE	**ADS-TS**	**EXP3-IX**	**EXP3++**	
Rank Avg	4.9	**2.6**	6.5	6.9	
Score Avg	95.57	97.39	94.11	93.56	

Proof of concept and result

We incorporated SLOPT and the aforementioned existing methods into AFL++ to compare the performance improvement (of code coverage).

The reasons we chose AFL++ as baseline

- It exactly has the mutation scheme we mentioned.
- It produces good results on various benchmarks such as FuzzBench.
- It is one of the fuzzers used in OSS-Fuzz
  - Our performance improvement can be immediately available in OSS-Fuzz.

Our result

- the best in the same metrics as FuzzBench (average score and rank),
- also better in statistical significance and effect size for each program.

PUT	AFL++ [21]	MOpt [40]	CMFuzz [64]	Karamcheti [35]	$Havoc_{MAB}$ [66]	SLOPT
…						
Rank Avg	4.4	*5.1*	3.9	2.7	3.0	**1.6**
Score Avg	92.81	*92.13*	96.37	97.59	95.69	**99.28**

Limitation and future work

SLOPT did not outperform the others in every program.

Estimated Cause
Considering that SLOPT uses a sole operator while the others mix operators, it may imply mixing them is fruitful sometimes.

Difficulty we faced in analysis
A lot of experiments would be needed to reveal when exactly the "sometimes" is.
Repeating experiments was very hard due to an enormous amount of required computing resources (more than thousands CPU days).

Possible future direction
Investigating the cause may produce a more efficient hybrid optimization method of mixing and not mixing operators.

Conclusion

To boost the performance of mutation-based fuzzer while keeping its generality, we proposed SLOPT, a general optimization framework employing bandit algorithm.

Although some existing methods share the same motivation and direction as ours, we made three observations they unconsidered and utilized the findings in ours.

The evaluation result showed SLOPT showed the best performance (code coverage) reliably on average. However, considering it was not the best always, there is still room for further investigation and improvement.

Thank you!

Page list for Q&A:

- Related Work: 7
- Observations#1: mutation scheme 10
- Observations#2: batch size 12
- Observations#3: optimization algorithm difference 15
- Result: 16
- Limitation: 17

Git Repo: https://github.com/RICSecLab/SLOPTAFLpp

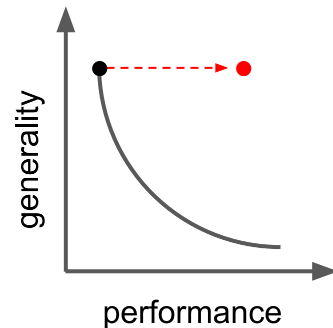
The followings are Omitted pages

Related work: online optimization for mutation

Some studies employed online optimization in mutation to boost fuzzing performance while keeping the generality.

Some existing methods employ bandit algorithms because optimizing mutation can be naturally interpreted as bandit problem.

- Karamcheti et al. used Thompson Sampling (a bandit algorithm) to optimize operators with the same motivation as MOPT.
- CMFuzz extended the algorithm of Karamcheti et al. with LinUCB (a contextual bandit algorithm).
- HavocMAB finally suggested online optimization with KL-UCB (a bandit algorithm) through experiments similar to MOPT.



Observations#2: batch size

Batch size is a parameter controlling how many operators applied to one input. We ran AFL++ and recorded batch size for each valuable input. Then, we found

- Batch size affects the number of valuable inputs produced.
- Which batch size is the most effective depends on programs.

Thus, batch size also should be an optimization target.

Algorithm 1 Conventional Random Mutation Scheme

Input: *seed* – a test case to be mutated
Output: *input* – a new input to be tested

```
function RandomMutation(seed)
    input ← CopyBytesFromSeed(seed)
    batch_size ← DecideBatchSize()
    for i ← 1 to batch_size do
        mutation ← SelectOperator()
        pos ← SelectPosition(input)
        input ← ApplyOperator(mutation, input, pos)
    end for
    return input
end function
```

Observations#2: batch size

While HavocMAB previously found this finding with a different observation,

our observation also found another phenomenon:

- Distributions of effective batch size are different depending on seed size.
- This means optimization should be done independently for different seed sizes.