Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Extending generator-based fuzzing with power schedules

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Iakov Evlakhov
geboren am: 30.1.2000
geboren in: Pochwistnewo

Gutachter/innen: Prof. Dr. rer. nat. Lars Grunske
Dr. Thomas Vogel

eingereicht am: ..... verteidigt am: .....

Abstract

While generator-based Fuzzing approach becomes more popular, it still has some room for improvement. While new test is generated by mutating the seed, it is added into the set of seeds, if it is considered to be interesting, or otherwise discarded, but once the seed is in the set, it never gets discarded. In fuzzing tools, which generate equal amounts of tests for each of the seed of the set, some seeds may be mutated often, even when they are not considered to be expected to generate new interesting tests.

Power Scheduling algorithms propose to adapt the strategy of what inputs to choose from the set and how often to mutate them. While most Power Scheduling algorithms were implemented and evaluated on mutation-based fuzzing tools, in this work, two of the Power Scheduling approaches are implemented into Zest, a generator-based Fuzzing approach, and evaluated against Zest baseline.

Contents

1 Introduction

Due to its efficiency, fuzzing has become one of the most successful vulnerability discovery techniques. Various fuzzing tools (fuzzers) can be classified by test case generation and type of program execution. [3]

There are two types of test case generation - mutation-based fuzzers take initial valid seed inputs and generate test cases by mutating those inputs. The most popular mutation-based fuzzer is AFL. [1] Generation-based tools like Zest [10] rely on domain-specific input generators to produce syntactically valid inputs. These input generators produce test inputs based on a sequence of different random choices that determine the syntactic structure and semantic elements of the input.

One of the problems of both types of fuzzing techniques can be demonstrated in the following algorithm of Zest, which other typical fuzzers like AFL also have. The issue can be demonstrated in Algorithm 1 in lines 5 and 12. Some inputs may be added to the queue and after some time of fuzzing those inputs may create "head of line blocking", if those inputs are unlikely to produce new interesting inputs or results. Worst-case scenario is if queue consists of hundreds of inputs and only last element of the queue is able to produce a new interesting input.

Algorithm 1 Coverage-guided fuzzing.

Input: program p, set of initial inputs I
Output: a set of test inputs and failing inputs

```
1:  S ← I
2:  F ← ∅
3:  totalCoverage ← ∅
4:  repeat
5:    for input in S do
6:      for 1 ≤ i ≤ NUMCANDIDATES(input) do
7:        candidate ← MUTATE(input, S)
8:        coverage, result ← RUN(p, candidate)
9:        if result = FAILURE then
10:         F ← F ∪ candidate
11:       else if coverage ⊈ totalCoverage then
12:         S ← S ∪ {candidate}
13:         totalCoverage ← totalCoverage ∪ coverage
14: until given time budget expires
15: return S, F
```

Figure 1: Fuzz testing process [10]

Power Schedules techniques allow fuzzer to solve this problem by reorganizing the order of those inputs and the amount of times each of the inputs from the queue is changed. Power Schedules have been implemented in many ways for non generator based fuzzers like AFL, e.g. [11], [6], [9]. To increase the efficiency of such fuzzing techniques, Power Schedules make it possible to focus on fuzzing only the "interesting" seeds, while other inputs are barely fuzzed. In this work, following Power Scheduling techniques are implemented into generator-based fuzzer Zest and evaluated against Zest baseline:

- AFLFast - Coverage-based Greybox Fuzzing as Markov Chain [7]
- Entropic - Boosting Fuzzer Efficiency: An Information Theoretic Perspective [5]

The main contributions of this works are:

- Implementation of 2 types of Power Schedules proposed in AFLFast and one proposed in Entropic.
- Evaluation of both implementations against the baseline Zest on the following criteria:
  - How efficient are the implemented Power Schedules in comparison to baseline Zest considering total branch coverage?
  - What overhead do the implemented Power Schedules produce?
  - How efficient are the implemented Power Schedules in comparison to baseline Zest considering total amount of failures found and amount and average times of each of the unique failures found?

The rest of this work is organized as follows. Chapter 2 gives a background on Zest and Power Schedules and the chosen algorithms. In chapter 3, the approach is presented in detail. The approach is then evaluated in chapter 4. At the end threats to validity is discussed in chapter 5.

2 Background

2.1 Power Schedules

The purpose of Power Schedules is to prioritize the seeds, which will be changed more, so that the head of line blocking does not happen and only interesting inputs are fuzzed often. The Power Scheduling technique can be split into two different parts - Search Strategy and Power Schedule itself. The goal of Power Scheduler is to decide which seed to produce children from, how often and at what point at a time.

2.1.1 Search Strategy

The search strategy decides the order in which seeds are chosen from the seed corpus. It can be the the other order computed after each completion of the queue or each seed can be chosen via sheduling techniques such as lottery ticket scheduler.

2.1.2 Power Schedule

The Power Schedule decides a seed's energy, which may decide how many inputs are generated by fuzzing the seed or which seed to choose more often from the queue. It is based on some criteria specified by the Power Schedule, e.g. number of executions, probability of discovering interesting input or the quality of the input.

2.2 Zest and JQF

Zest is generator-based Coverage-Guided Fuzzing tool. Coverage-based greybox fuzzing such as AFL [1] or libFuzzer approach testing via light code instrumentation. CFG operates by mutating the existing inputs via operations like bit flips or byte-level splicing in order to produce new inputs. If the new inputs lead to new coverage in the program they are saved in the queue for further mutation. The problem with mutation-based CFG algorithms is that it takes long to discover deeper bugs, which make it unsuitable for testing with limited time.

On the other hand, generator-based testing tools like Zest or QuickCheck [8] generate syntactically valid inputs. In comparison to QuickCheck, Zest converts QuickCheck-like random input generators into parametric generators. Bit-level mutations then represent the structural mutations in the space of syntactically valid inputs. This approach is combined with CGF, in order to guide the test-input generator towards semantic validity and increased coverage in not only syntactical but also in semantic analysis stages.

The feedback from each test is then utilized in form of coverage counts to determine, whether the input is being saved into the queue. If input discovered new coverage, then the input is saved in the seeds queue. More than that, Zest also saves inputs, which lead towards new valid coverage, as it can be seen in Algorithm 2 in the lines 15-17 and 18-20.

Algorithm 2 The Zest algorithm, pairing parametric generators with feedback-directed parameter search. Additions to Algorithm 1 highlighted in grey.

Input: program p, generator q
Output: a set of test inputs and failing inputs

1: $\mathcal{S} \leftarrow$ {RANDOM}
2: $\mathcal{F} \leftarrow \emptyset$
3: $totalCoverage \leftarrow \emptyset$
4: $validCoverage \leftarrow \emptyset$
5: $g \leftarrow$ MAKEPARAMETRIC(q)
6: **repeat**
7: **for** $param$ in $\mathcal{S}$ **do**
8: **for** $1 \leq i \leq$ NUMCANDIDATES($param$) **do**
9: $candidate \leftarrow$ MUTATE($param$, $\mathcal{S}$)
10: $input \leftarrow g(candidate)$
11: $coverage, result \leftarrow$ RUN(p, $input$)
12: **if** $result$ = FAILURE **then**
13: $\mathcal{F} \leftarrow \mathcal{F} \cup candidate$
14: **else**
15: **if** $coverage \not\subseteq totalCoverage$ **then**
16: $\mathcal{S} \leftarrow \mathcal{S} \cup \{candidate\}$
17: $totalCoverage \leftarrow totalCoverage \cup coverage$
18: **if** $result$ = VALID and $coverage \not\subseteq validCoverage$ **then**
19: $\mathcal{S} \leftarrow \mathcal{S} \cup \{candidate\}$
20: $validCoverage \leftarrow validCoverage \cup coverage$
21: **until** given time budget expires
22: **return** $g(\mathcal{S})$, $g(\mathcal{F})$

Figure 2: Fuzz testing process [10]

Each seed of the queue, as it can be seen in line 8, is fuzzed a set amount of times. Baseline is the equal for all the seeds, but some seeds are favored and therefor mutated more, if those have higher coverage than the ones before, e.g. for inputs A and B if $valid\_coverage(A) \subseteq valid\_coverage(B)$, then B is prioritized and fuzzed more often that A.

Zest is implemented in JQF [2] as a guidance. JQF is a feedback-directed fuzz testing platform for Java which enables the writing of the guidances for coverage-guided fuzzing algorithms such as Zest. It instruments the code of the program, which allows guidances to use information like results of the input executed by a program and branch coverage.

2.3 AFLFast

2.3.1 Theory

In Coverage-based Greybox Fuzzing as Markov Chain [7], fuzzing algorithm is modeled as Markov chain. The key idea is that the whole search space can be seen as Markov Chain, while inputs exercising different paths can lead to different paths via mutations of the inputs. All the possible paths are then states and each of the possible mutations of the input exercising path I leading to the new Input exercising other path J can be seen as a transition probability between I and J.

Initially, state space is defined by the initial seeds, if any are present. During fuzzing

time, the amount of stats is increased via discovering new paths. As more test inputs are generated, less paths are considered to be interesting. That is why path discovery decelerates monotonically. As the number of discovered paths approaches the total number of paths, distribution of the current Markov chain approaches the stationary distribution of Markov chain of the whole program.

2.3.2 Power Schedule and Search Strategy

The objective of AFLFast is to discover interesting paths, such that those are not exercised by the inputs saved in the queue, while generating minimal amount of inputs. That is achieved via prioritizing low-frequency paths over high-frequency paths.

The energy of each state (each path) determines the number of inputs which should be generated from a seed exercising the path, when the path is chosen from the queue. The energy is calculated by the Power Schedule formula.

Two relevant formulas for this work from AFLFast are:

- exponential schedule (FAST)
- quadratic schedule (QUAD)

Exponential schedule (FAST) Energy **p(i)** for a state (path) **i** is calculated via following formula:

$p(i) = min(\frac{a(i)}{b(i)} \cdot \frac{2^{s(i)}}{f(i)}, M)$,

with $\frac{a(i)}{b(i)}$ being the fuzzers standard energy, standard amount of children produced from the input exercidsing path **i**, **s(i)** being the amount of times, the input, exercising path **i**, has been chosen from the queue before, **f(i)** amount of fuzz exercising path **i**, e.g. amount of inputs, produced during fuzzing campaign, which led to the path **i**, and M being the upper bound for the energy.

Quadratic schedule (QUAD) Energy **p(i)** for a state (path) **i** is calculated via following formula:

$p(i) = min(\frac{a(i)}{b(i)} \cdot \frac{s(i)^i}{f(i)}, M)$,

with $\frac{a(i)}{b(i)}$ being the fuzzers standard energy, standard amount of children produced from the input exercidsing path **i**, **s(i)** being the amount of times, the input, exercising path **i**, has been chosen from the queue before, **f(i)** amount of fuzz exercising path **i**, e.g. amount of inputs, produced during fuzzing campaign, which led to the path **i**, and M being the upper bound for the energy.

Upper bound **M** is specified to be 160,000 in AFLFast.

Search strategy determines, which seeds have to be fuzzed next. AFLFast prioritizes small **s(i)**, meaning, the inputs, which exercise the path, with a smallest number of time that the path has been chosen from the queue. If there are several inputs, which exercise the paths, which has been chosen the same amount of times before, then the

path with the least amount of fuzz **f(i)** is chosen. Search strategy does not effectively change anything except for that the more interesting paths are chosen first.

2.4 Entropic

2.4.1 Theory

Boosting Fuzzer Efficiency: An Information Theoretic Perspective [5] (Entropic) is based on the paper STADS: Software Testing as Species Discovery [4] and utilizes principles of Shannon's entropy [12]. Fuzzing campaign can be seen as discovery of species, which are defined in this work as each branch exercised by an input. That means, if multiple inputs exercise same branches, then the branch belongs to both of the inputs. The species discovery is therefor used as the increase of the branch coverage. Fuzzing campaign starts with zero species discovered and while the new inputs are produces, feedback of the program gives information about whether new species have been discovered.

Shannon's entropy measures the average amount of information of each Input about the species of the program. If there are **S** distinct species, then entropy **H** is: $H = -\sum_{i=1}^{S} p_i log(p_i)$, while p_i is defined as the probability that the **n**-th generated Input X_n belongs to the species D_i, $p_i = P[X_n \in D_i]$

Shannon's entropy is defined for the distribution, where each input belongs to exactly one species. That's why it is proposed in [5] to normalize the probabilities, such that: $H = -\sum_{i=1}^{S} p_i' log(p_i') = log(\sum_{j=1}^{S} p_j) - \frac{\sum_{i=1}^{S} p_i log(p_i)}{\sum_{j=1}^{S} p_j}$, where $p_i' = p_i / \sum_{j=1}^{S} p_j$

The Local Entropy of a Seed **t** is then :

$H^t = log(\sum_{j=1}^{S} p_j^t) - \frac{\sum_{i=1}^{S} p_i^t log(p_i^t)}{\sum_{j=1}^{S} p_j^t}$

According to theorem 1 from [5], which can be seen in Figure 3

THEOREM 1. *Let Shannon's entropy be defined as in Equation (12). Let $\Delta(n)$ be the expected number of new species the fuzzer discovers with the $(n+1)$-th generated test input, then*

$$H = \log(c) + \sum_{n=1}^{\infty} \frac{\Delta(n)}{cn} \tag{14}$$

characterizes the rate at which species are discovered in an infinitely long-running campaign, where $c = \sum_{j=1}^{S} p_j$ is a normalizing constant.

Figure 3: Theorem 1

Entropy measures the species discovery rate Δ(n) over an infinitely long-running fuzzing campaign where discovery rate decreases as the number of tests **n** goes to infinity.

Shannon's entropy **H** is the estimated on how often each species has been observed during fuzzing campaign. Incidence frequency Y_i for species D_i is defined as the number of generated inputs that belong to D_i. For undiscovered species incidence is zero, Y_i

$= 0$. Using the incidence frequency, entropy can then be estimated using maximum likelihood estimator, which gets to the following formula:

$H = log(\sum_{j=1}^{S} Y_j) - \frac{\sum_{i=1}^{S} Y_i log(Y_i)}{\sum_{j=1}^{S} Y_j}$

2.4.2 Power Schedule and Search Strategy

From the theory, [5] suggests using Shannon's local entropy of each Input as the energy. More than that, since every new seed **t** will be assigned zero energy at first $H^t = 0$, they suggest adapting the formula using Laplace estimator:

$p^t = \frac{Y_i^t + 1}{S_g + \sum_{j=1}^{S} Y_j^t}$, where $S_g = S(n)$ is the number of globally discovered species.

Therefor, energy H is then:

$H^t = log(S_g + \sum_{j=1}^{S} Y_j) - \frac{\sum_{i=1}^{S} (Y_i+1) log(Y_y+1)}{S_g + \sum_{j+1}^{S} Y_j}$

Entropic Algorithm can be seen in 4. In the lines 2-4 each of the inputs is assigned the energy using the previous formula, which is then normalized in the line 4. After that search strategy is applied, which makes each Input **t** get randomly chosen from the queue with the probability of the normalized energy of the seed. After mutating the input and executing it, local incidences are changed using the covered coverages gathered from the feedback of the program, which can be observed in the lines 9-11.

4

Algorithm 1 ENTROPIC Algorithm.

Input: Program $\mathcal{P}$, Initial Seed Corpus C

1: **while** ¬TIMEOUT() **do**
2: **for all** $t \in C$. ASSIGNENERGY(t) *// power schedule*
3: $total = \sum_{t \in C} t.energy$ *// normalizing constant*
4: **for all** $t \in C$. $t.energy = \frac{t.energy}{total}$ *// normalized energy*
5: t = sample t from C with probability $t.energy$
6: t' = MUTATE(t) *// fuzzing*
7: **if** $\mathcal{P}(t')$ crashes **then return** crashing seed t'
8: **else if** $\mathcal{P}(t')$ increases coverage **then** add t' to C
9: **for all** covered elements $i \in \mathcal{P}$ exercised by t' **do**
10: $Y_i^t = Y_i^t + 1$ *// local incidence freq.*
11: **end for**
12: **end while**
return Augmented Seed Corpus C

Figure 4: Entropic Algorithm

It is also proposed to use abundance threshold to only measure information about rare species, which makes only the rare species, meaning species which have local incidence lower than the threshold, be relevant to the energy calculation. Proposed thresholds are **0x100, 0x1000, 0x10000**

Search strategy determines, which seeds have to be fuzzed next. Entropic chooses an input randomly with the probabilities of the input being chosen being proportional to their energy.

3 Approach

The general approach of this work will be focused on the implementation aspect of this work.

As mentioned in 2.2, Zest has been implemented in JQF. For this work, following files were needed to be modified for each version of the Power Scheduling extension:

- JQF/bin/jqf-zest - this bash file starts the JQF with the Guidance Zest with all the relevant parameters like classpath, test class, test method, output directory and seed files.
- JQF/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestCLI.java - this class enables the command line interface for the JQF with the Guidance Zest.
- JQF/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestDriver.java - the driver gets the parameters from two files mentioned before and starts the zest guidance, while checking, if parameters satisfy the zest guidance.
- JQF/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestGuidance.java - the guidance is the java class, where the fuzzing algorithm is implemented.

For each of the following algorithms, each of those files has been adapted without changes in the functionality. Only Power Scheduling algorithms have been implemented to ensure the validity of the experiments. Only non Power Scheduling part, which has been edited is console output, the reasoning for which will be explained further.

At the end, there are 5 versions of the earlier mentioned files, which allows to use them independently from each other. The implementation can be found under:
`https://github.com/Grifon321/JQF-Zest-Power-Scheduler`
For AFLFast Exponential schedule (FAST):

- JQF-Zest-Power-Scheduler/bin/jqf-zest-AFL
- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestAflCLI.java
- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestAflDriver.java
- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestAflGuidance.java

For AFLFast Quadratic schedule (Quad):

- JQF-Zest-Power-Scheduler/bin/jqf-zest-Quad
- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestAflQuadCLI.java

- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestAflQuadDriver.java
- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestAflQuadGuidance.java

For Entropic:

- JQF-Zest-Power-Scheduler/bin/jqf-zest-Entropic
- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestEntropicCLI.java
- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestEntropicDriver.java
- JQF-Zest-Power-Scheduler/fuzz/src/main/java/edu/berkeley/cs/jqf/fuzz/ei/ZestEntropicGuidance.java

3.1 AFLFast

To implement AFLFast algorithm from Section 2.3, jqf-zest-AFL, ZestAflCLI.java and ZestAflDriver.java have only been modified, so that the ZestAflQuidance.java can be launched via both CLI and bash script. The most important changes have been done int ZestAflQuidance.java. Same holds for Quad version.

Both algorithms FAST and QUAD from section 2.3 are the same, with the exception, that only the formula is different.

The following Figure 5 shows the difference to the initial Zest algorithm, while marked text represents the changed or newly added lines.

In the lines 2-5 of 5 all the variables are initialized. That can be found in lines 256-265 and 278 in ZestAflGuidance. While the search strategy is different from Zest baseline, the lines 11-13 of 5 also varies from the baseline, the exact queue order can be found in 29.

The implementation of the search strategy can be found in the lines 284-318. In the function findOrderOfExecution(), it is determined, in which order seeds will be fuzzed more, depending on primarily amount of times input was chosen from the queue **s(i)**- the smallest number is preferred, and the energy **f(i)**. The inputs, which are added to the savedInputs queue are fuzzed in the next cycle of the for loop (lines 755-761)

Initially, each of the fuzzed seeds is granted the base amount of children, specified by Zest - 20. Once the amount of the children inputs is exceeded, next input from the queue is chosen.

After a new input is executed, the feedback is used to increment the amount of times each of the branches has been discovered (lines 18 and 26 in 5), if the coverage of the input is already saved in one of the coverages of the inputs in the queue. If the input is saved as new interesting input, then the AnountOfTimesFuzzed (**fuzzedToCoverageCounter** in the implementation) of this new input is set to 1, since this is the only input, which

```
   Input: program p, generator q
   Output: a set of test inputs and failing inputs
 1 SavedInputs ⇐ Random
 2 QueueOrder ⇐ [0]
 3 EnergyOfInputs ⇐ [standardEnergy]
 4 AmountOfTimesFuzzed ⇐ [0]
 5 NumberOfSamePaths ⇐ [0]
 6 Failures ⇐ ∅
 7 totalCoverage ⇐ ∅
 8 validCoverage ⇐ ∅
 9 parametricGenerator ⇐ makeParametric(q)
10 while given time budget is not expired do
11     for paramIndex in QueueOrder do
12         for 1 ≤ i ≤ numCandidates( SavedInputs(paramIndex) ) do
13             candidate ⇐ mutate( SavedInputs(paramIndex), S)
14             input ⇐ parametricGenerator(candidate)
15             coverage, result ⇐ run(p, input)
16             if result = Failure then
17                 Failures ⇐ Failures ∪ candidate
18                 UpdateNumberOfSamePaths()
19             else
20                 if coverage ⊈ totalCoverage then
21                     SavedInputs ⇐ SavedInputs ∪ {Candidate}
22                     totalCoverage ⇐ totalCoverage ∪ coverage
23                 if result = Valid and coverage ⊈ validCoverage then
24                     SavedInputs ⇐ SavedInputs ∪ {Candidate}
25                     validCoverage ⇐ validCoverage ∪ coverage
26                 UpdateNumberOfSamePaths()
27         AmountOfTimesFuzzed[paramIndex]++
28     assignEnergyInput(paramIndex)
29     QueueOrder ⇐ makeNewQueueOrder()
   return: g(SavedInputs), g(Failures)
```

Figure 5: Pseudocode of AFLFast implemented into Zest Guidance

led towards this unique coverage. Otherwise, Zest would have saved this input already (lines 830-862). That is done in the lines 863-879.

When the number of children exceeds the one which has to be generated, the amount of times input has been chosen from the queue is incremented and the next input starts to getting mutated. And the new energy is calculated.

After the cycle is done, new order of inputs execution is calculated based on energy, with the new inputs added from the last queue being fuzzed before the previous ones.

For FAST formula from Section 2.3.2 is used $p(i) = min(\frac{a(i)}{b(i)} \cdot \frac{2^{s(i)}}{f(i)}, M)$,

with the adjustment, that the upper bound of energy is equal to 1000 = Zest base amount of children generated (20) * Zest multiplier for prefered inputs (50).

The parameter is chosen that way due to the 1-hour to 3-hour runs, which have shown, that on average such upper bound shows the best results. Other parameters, which have been tested are:

- $0.5 \cdot 50 \cdot 20$

- $5 \cdot 50 \cdot 20$
- $10 \cdot 50 \cdot 20$
- $25 \cdot 50 \cdot 20$
- $50 \cdot 50 \cdot 20$

Except for Power Scheduling, only console has been changed, since the favored function briefly explained in Section 2.2 from Zest baseline is no longer needed and prevents the normal functionality of the implemented algorithms.

3.2 Entropic

To implement Entropic algorithm from Section 2.4 jqf-zest-Entropic, ZestEntropicCLI.java and ZestEntropicDriver.java have been modified in a such way, that the user can specifiy the threshold used in algorithm as well as additional constructors in ZestEntropicQuidance.java.

The following Figure 6 shows the difference to the initial Zest algorithm, while marked text represents the changed or newly added lines.

In the lines 7-9 of Figure 6 the energy is calculated, normalized and assigned to each input. The implementation can be found in the file ZestEntropicGuidance.java in the lines 141-180. The energy is calculated using local entropy of a seed from 2.5: $H^t = log(S_g + \sum_{j=1}^{S} Y_j) - \frac{\sum_{i=1}^{S}(Y_i+1)log(Y_y+1)}{S_g+\sum_{j+1}^{S} Y_j}$, where S_g is the number of globally discovered species and Y_i is the counter of the incidence of each branch covered by the input. As proposed in the base work, the threshold has been also implemented, where only branches with the local incidence less or equal than the threshold influence the amount of the energy. It is done via ignoring incidences with higher than threshold appearance counters (line 152 in ZestEntropicGuidance.java).

According to algorithm with the threshold, once all coverages achieve the threshold, no local incidences influence the Power Scheduling algorithm. Therefore, once all local incidences achieve the threshold, the threshold is set 2 times higher, which has not been discussed in the Entropic [5] (lines 162-165, 136-138 in ZestEntropicGuidance.java).

After each input is assigned energy, lottery ticket scheduler decides, which of the inputs will be fuzzed in the next iteration. Lottery ticket scheduler assigns each of the inputs an amount of tickets, proportional to the energy, and randomly chooses one ticket from the total amount. The input with the ticket is chosen for the next iteration.The implementation can be found in the lines 197-223 of ZestEntropicGuidance.java.

Then guidance works as before, with the exception, that the feedback of the program execution is also used to calculate local incidences of each of the inputs. It is done via incrementing the counters of each of the branches, which have been during the execution of the current input. It is done both for successful and failed results in the lines 16 and 19 in 6 (lines 933 and 976 in ZestEntropicGuidance.java).

As the algorithm 4 from [5] suggests, each of the newer inputs should be only mutated and executed once, while in this work small tests of 1-3 hours run have shown, that

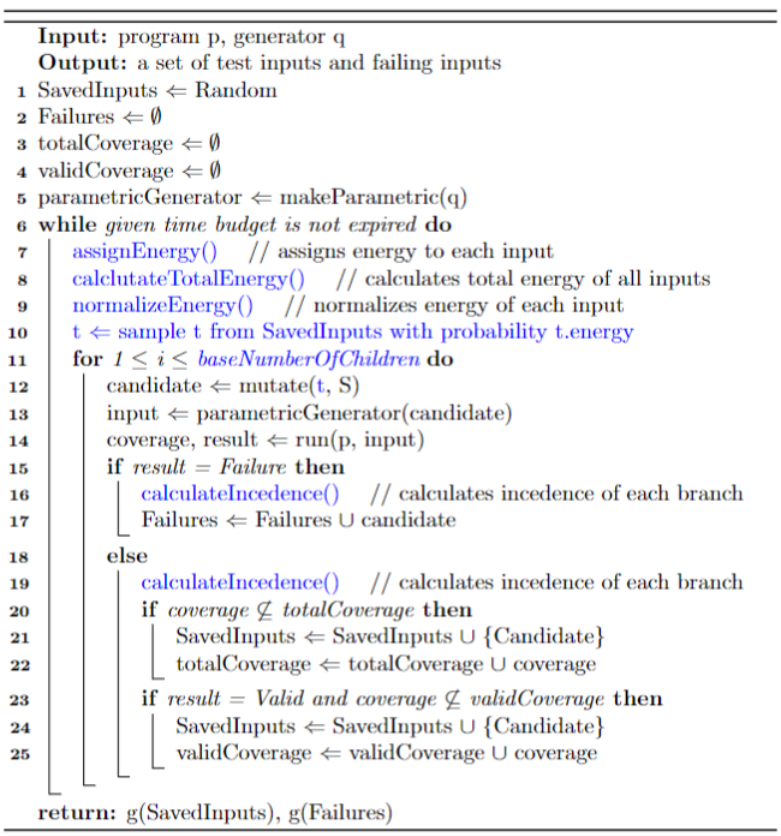

Figure 6: Pseudocode of Entropic implemented into Zest Guidance

the overhead slows the fuzzing algorithm, that is why in the line 11 the base amount of children from Zest is produced, before new input is chosen.

4 Evaluation

4.1 Experimental setup

In this section, each of the implementations of Power Scheduling algorithms will be tested against Zest baseline on the following criteria:

- How efficient are the implemented Power Schedulers in comparison to baseline Zest considering total branch coverage?
- What overhead do the implemented Power Schedulers produce?
- How efficient are the implemented Power Schedulers in comparison to baseline Zest considering total amount of failures found and amount and average times of each of the unique failures found?

Benchmarks. Baseline Zest as well as AFLFast Fast, AFLFast Quad and 2 versions of Entropic with the different thresholds (0x100 and 0x1000) have been tested on 5 benchmarks, proposed in Zest [10]:

- Apache Maven (Mvn) (99k LoC) - The test reads a pom.xml file and converts it into an internal Model structure. An input is valid if it is a valid XML document which conforms to the POM schema.
- Apache Ant (Ant) (223k LoC) - The test reads a build.xml file and populates a Project object. An input is valid if it is a valid XML document which conforms to the schema expected by Ant.
- Google Closure (Clo) (247k LoC) - The test statically optimizes JavaScript code and checks, whether it is suitable for the compiler. An input is valid if Closure returns without error.
- Mozilla Rhino (Rhii) (89k LoC) - The test compiles JavaScript to Java bytecode. An input is valid if Rhino returns a compiled script.
- Apache's Bytecode Engineering Library (BCE) (61k LoC) - The test parses and verifies Java bytecode. An input is valid if BCEL finds no errors up to Pass 3A verification.

Experimental Setup. Such design decisions have been made:

- **Duration**: As AFLFast and Entropic both propose different times, and as different works like [**?**] propose for the tests to be between 1 and 24 hours with popular choices being 1 hour, 6 hours and 24 hours, the duration was decided to be 6 hours, since some of the experiments were done for 6 hours in both AFLFast and Entropic.

- **Repetitions**: Each of the tests has been run 20 times due to the non-deterministic nature of fuzzing.
- **Generators**: Generators, which are publicly made available in Zest, has been used in this work to test each of the benchmarks.

All the experiments have been done on a server with the following specs:

- Modell: Dell R7515
- CPU: AMD EPYC 7713P
- CPUs/Cores/Threads: 1 / 64 / 128
- Takt: 2Ghz
- Ram: 256GB

During the Experiments, server has exclusively been used for this work.
All of the results can be found in under following link in the zip file named Results.zip:
`https://drive.google.com/drive/folders/1fsLgyrnavz9VL-F5-UpbelJ33tGfOrmC`
Since the zip file is 1.76GB, it could not fit into github, therefor drive.google.com has been chosen for that purpose.
Bash file used to run experiments can be found under following link:
`https://github.com/Grifon321/JQF-Zest-Power-Scheduler/tree/master/Tests`

4.2 Coverage

In this section, total branch coverage will be used as metric. All versions of Zest with Power Scheduling algorithms - FAST from AFLFast, QUAD from AFLFast and Entropic with thesholds 256 and 4096 will be compared against Zest baseline.

In the Figure 7 the average over all runs coverage achieved by all of the programs over all runs is shown for the benchmark Ant: Zest and Entropic4096 have exactly the same average coverage at all times. Entropic256 is slightly worse in the beginning but achieves insignificantly higher coverage after 2 hours of the run.

AFLFast and Quad have worse results, at 1 hour point both of the extensions achieve the coverage of baseline Zest from 10 minutes of fuzzing.

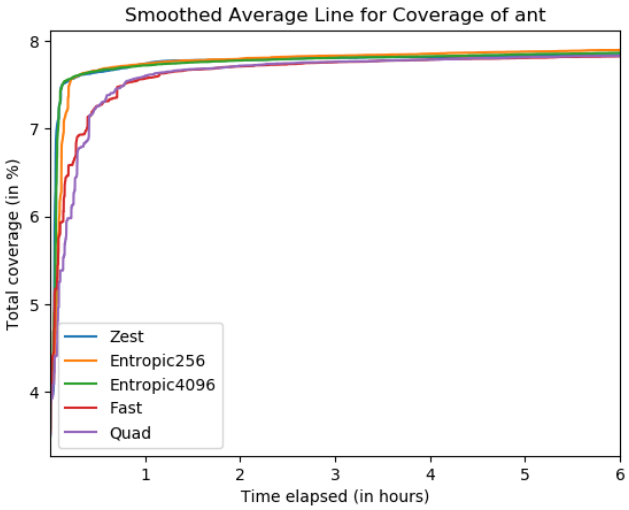

Figure 7: Ant Coverage over time

In the Figure 8 the average over all runs coverage achieved by all of the programs over all runs is shown for the benchmark BCE: Zest outperforms Entropic4096 and Entropic256 in the first 3 hours, while outperforming FAST and QUAD for the whole duration of the 6 hour runs.

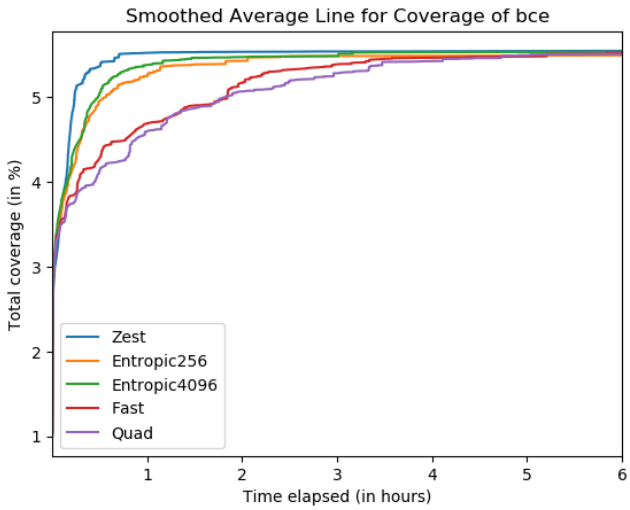

Figure 8: BCE Coverage over time

In the Figure 9 the average over all runs coverage achieved by all of the programs over all runs is shown for the benchmark Maven: As it can be seen, Zest achieves on average higher coverage in one hour than Entropic256 and Entropic4096 in 6 hour runs. FAST and QUAD can not even get to the coverage of Zest, which was achieved in 30 mins.

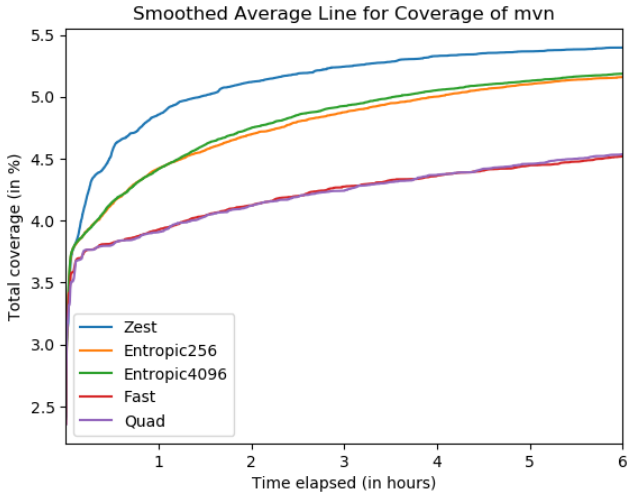

Figure 9: Mvn Coverage over time

In the Figure 10 the average over all runs coverage achieved by all of the programs over all runs is shown for the benchmark Closure: Zest outperforms other versions by 10% after 1.5 hours of fuzzing.

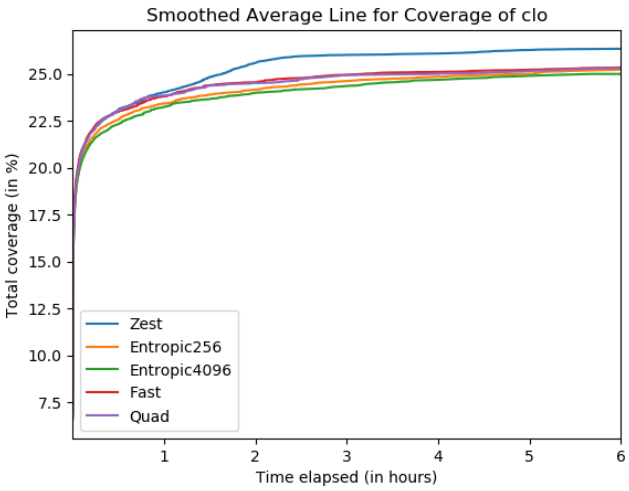

Figure 10: Clo Coverage over time

In the Figure 11 the average over all runs coverage achieved by all of the programs over all runs is shown for the benchmark Rhino: All the extensions achieve almost the same coverage at all times.

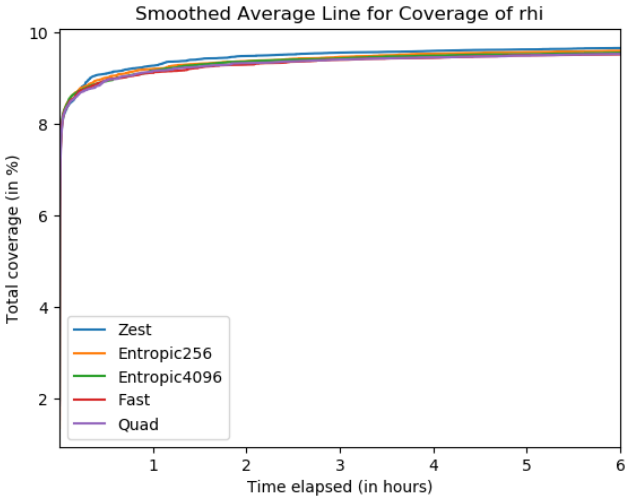

Figure 11: RHINO Coverage over time

To summarize, FAST and QUAD extensions provide much lower coverage, while both versions of Entropic provide slightly lower total coverage in comparison to Zest, except for Apache Maven benchmark. In Apache Maven both versions of Entropic only achieve only 90% of the Zest's coverage in the end of the 6 hour tests. Based on total coverage as metric, Entropic version with the threshold 4096 slightly outperforms the version with the threshold 256.

4.3 Overhead

In this section, amount of inputs executed by each of the extension will be discussed.

In the Figure 12 the average over all runs amount of total inputs executed by all of the programs is shown for the benchmark Ant: As it can be seen, the total amount of inputs of FAST and QUAD largely exceeds the amount of total inputs executed by Zest, Entropic256 and Entropic4096.

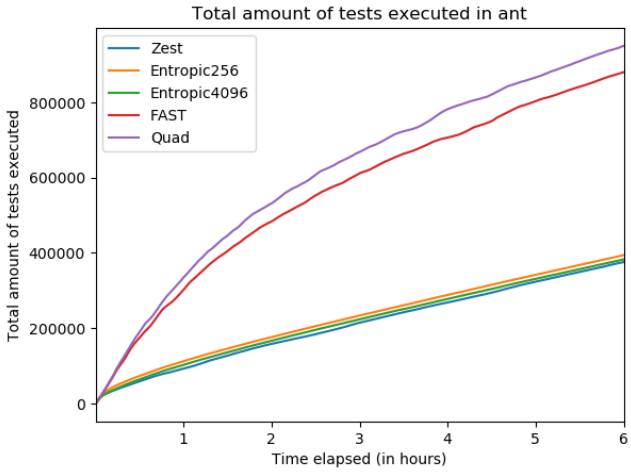

Figure 12: Amount of total inputs executed over time in Ant

In the Figure 13 the average over all runs amount of valid inputs executed by all the programs is shown for benchmark Ant:

Zest generates the biggest amount of valid inputs, while Entropic256 and Entropic4096 generate around 75% of valid inputs of Zest baseline. FAST and QUAD generate only around 60% of the valid inputs.

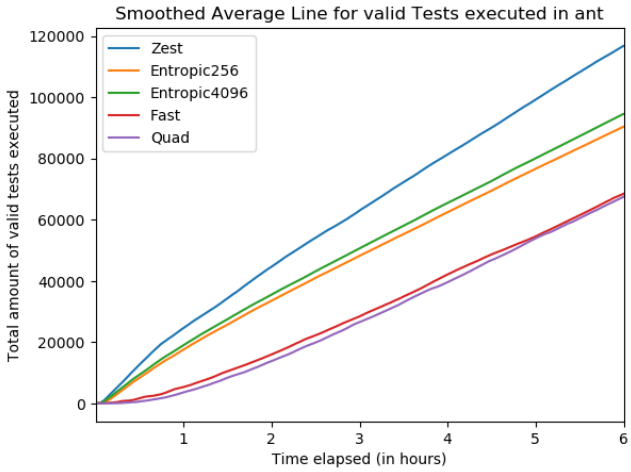

Figure 13: Amount of valid inputs executed over time inAnt

In the Figure 14 the average over all runs amount of total inputs executed by all of the programs is shown for the benchmark BCE: As it can be seen, the total amount of inputs of FAST and QUAD exceeds the amount of total inputs executed by Zest, Entropic256 and Entropic4096 by roughly 30%.

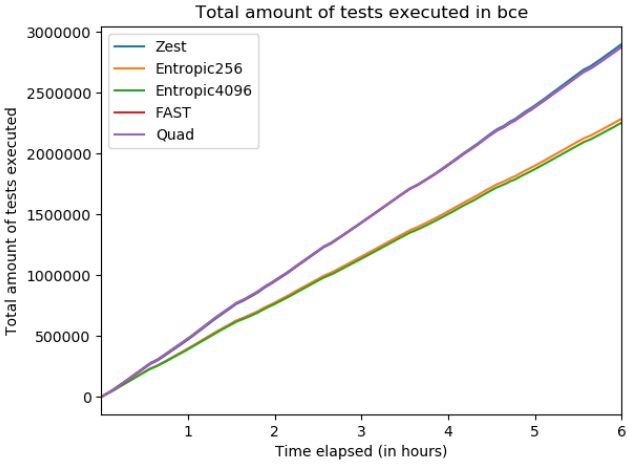

Figure 14: Amount of total inputs executed over time in BCE

In the Figure 15 the average over all runs amount of valid inputs executed by all the programs is shown for benchmark BCE:

Zest generates the biggest amount of valid inputs, while other versions with all Power Scheduling algorithms generate same amount of inputs.

Figure 15: Amount of valid inputs executed over time in BCE

In the Figure 16 the average over all runs amount of total inputs executed by all of the programs is shown for the benchmark Maven: Zest generates the highest amount of total inputs, followed by FAST and QUAD extensions with roughly 10% less generated tests and by Entropic256 and Entropic4096 by roughly 15%.

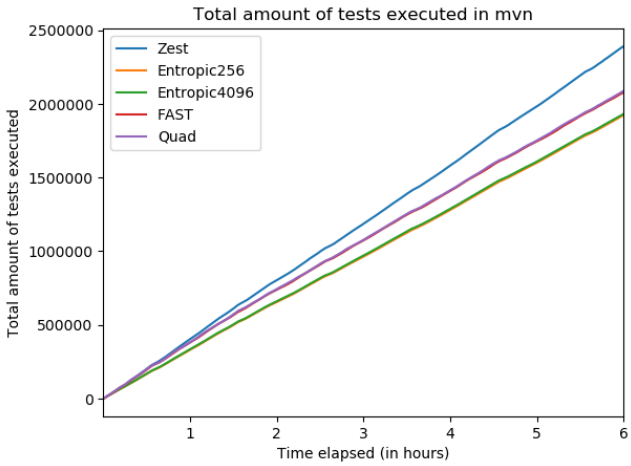

Figure 16: Amount of total inputs executed over time in Mvn

In the Figure 17 the average over all runs amount of valid inputs executed by all the programs is shown for benchmark Maven:

Zest generates the biggest amount of valid inputs, while Entropic256 and Entropic4096 generate around 50% of the valid inputs from zest and FAST and QUAD generate around 40%.

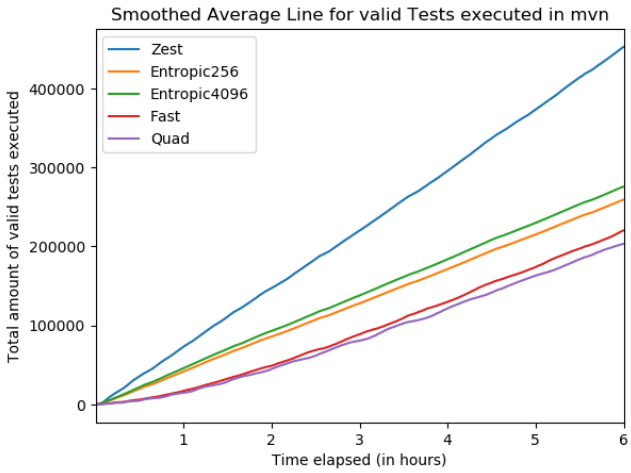

Figure 17: Amount of valid inputs executed over time in Mvn

In the Figure 18 the average over all runs amount of total inputs executed by all of the programs is shown for the benchmark Closure:

Zest, FAST and QUAD generate roughly the same amount of total inputs, while Entropic256 and Entropic4096 generate around 30% of other versions.

Figure 18: Amount of total inputs executed over time in Clo

In the Figure 19 the average over all runs amount of valid inputs executed by all the programs is shown for benchmark Closure:

Zest generates the highest amount of valid inputs, while FAST and QUAG generate around 70% of valid inputs and Entropic256 and Entopic4096 around 30%.

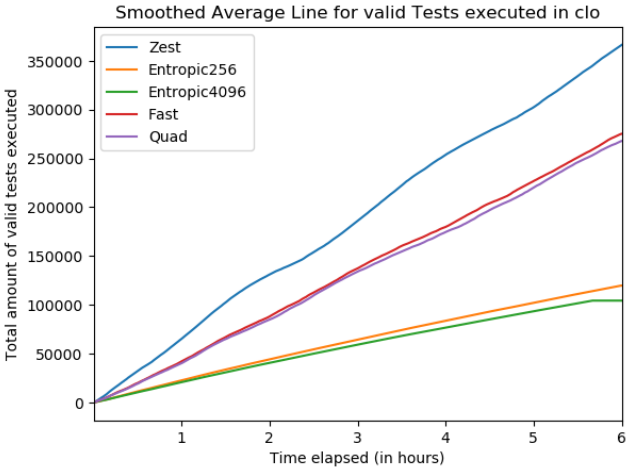

Figure 19: Amount of valid inputs executed over time in Clo

In the Figure 20 the average over all runs amount of total inputs executed by all of the programs is shown for the benchmark Rhino:

Zest, FAST and QUAD generate roughly the same amount of total inputs, while Entropic256 and Entropic4096 generate around 65% of other versions.

Figure 20: Amount of total inputs executed over time in Rhi

In the Figure 21 the average over all runs amount of valid inputs executed by all the programs is shown for benchmark Rhino:

Zest generates the highest amount of valid inputs, followed by FAST and QUAD with roughly 10% less valid inputs and Entropic256 and Entopic4096 with 50% of the inputs generated by Zest baseline.

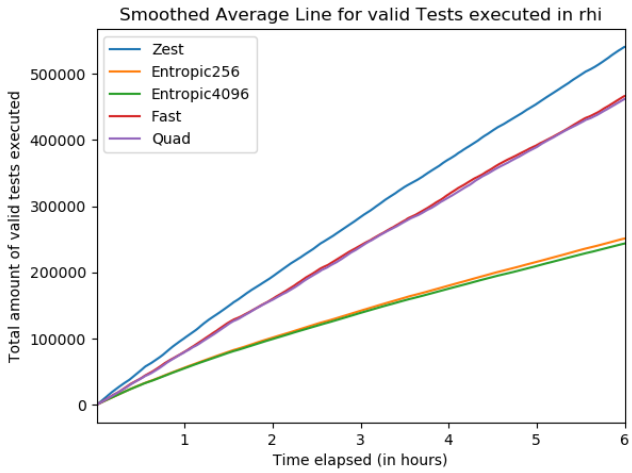

Figure 21: Amount of valid inputs executed over time in Rhi

To summarize, FAST and QUAD extensions produce slightly lower amount of total tests as Zest except for Apache Ant benchmark. In that benchmark FAST and QUAD generate almost 3 times more tests, as it can be seen in Figure 12 while producing way less valid tests as it can be seen in Figure 13. In other benchmarks, total amount of tests is varies from 80% to 95% of the amount of total test from Zest baseline.

In Ant that means, way more invalid tests are generated in the first half of the runs, while more valid tests are produced after 3 hour mark. The difference in the amount

of total inputs executed can be explained by invalid tests requiring way less time to fuzz and invalid tests being prioritized more by FAST and QUAD than from Zest baseline. Amount of total tests executed versus amount of valid tests executed shows same behaviour. Overhead of FAST and QUAD is almost the same.

Total and valid amounts of test executed by Entropic versions with thresholds 256 and 4096 are nearly identical to each other, while Entropic with the threshold 256 shows slightly better results.

In Apache Ant, Entropic versions generate roughly the same amount of total inputs, while the amount of valid inputs generated is 80% from Zest baseline,which shows, that Entropic version still prioritized invalid inputs more, which are executed quicker than valid ones.

In benchmarks BCEL, Maven and Rhino, Entropic versions generated 85%, 75% and 60%, while in Closure and Rhino only 35% of total tests were executed, while proportions between the total amount of executed tests and valid amount of executed tests remains scewed towards slightly lower amount of valid tests.

4.4 Unique Failures

Another metric, which can be utilized to understand, whether the power scheduling algorithms improves the efficiency, is the amount of unique failures, how often and how fast they were found. Zest saves all the failures, so the deduplication is needed. Therefore in this work the failures with a unique combination of a type of failure and 3 latest stack traces have been used to get rid of the duplicates.

In the following tables types of failures are shown as well as the average amount of time used by each of the programs under test on pair with the number of fuzzing repetitions, which led to exact same failure with the latest 3 stack traces.

Ant: Results of Ant can be seen in the following table 1: Power Schedules FAST and QUAD from AFLFast lead towards significant slowdown by about 5-6 times, while Entropic with the threshold 4096 is almost as good as baseline Zest and Entropic with the threshold 256 leads towards decrease of efficiency.

Failure	Zest	FAST	QUAD	Entropic256	Entropic4096
IllegalState Exception	301s (20)	1619s (20)	1918s (20)	483s (20)	338s (20)

Table 1: Average time to find each of the unique bugs found in Ant

BCE: Results of BCE can be seen in the following table 2: Power Schedules FAST and QUAD from AFLFast lead towards significant slowdown by about 2-5 times, while finding some failures rarer, e.g. AssertionViolatedException 1-3. Entropic with the threshold 4096 shows worse results than Zest in comparison to Ant with the average

time required to find a failure increases by 1.5-3 times with the exception of ClassFormatException 2. Entropic with the threshold 256 performs better than Zest on ClassFormatExceptions with insignificant speedup and worse on AssertionViolatedExceptions with 1.5 slowdown.

Failure	Zest	FAST	QUAD	Entropic256	Entropic4096
Assertion Violated Exception 1	2452s (14)	12673s (11)	10129s (10)	3858s (13)	4269s (12)
Assertion Violated Exception 2	2746s (13)	10079s (8)	9520s (9)	4936s (15)	7822s (9)
Assertion Violated Exception 3	4950s (12)	9790s (8)	12054s (9)	7070s (16)	6055s (14)
ClassFormat Exception 1	3876s (20)	7077s (18)	7115s (19)	3212s (20)	5051s (20)
ClassFormat Exception 2	230s (20)	378s (20)	240s (20)	210s (20)	155s (20)

Table 2: Average time to find each of the unique bugs found in BCE

Maven: No exceptions have been found in apache maven in 6 hours runs.

Closure: Results of Clo can be seen in the following table 3: NullpointerException has been found by all the versions at approximately the same time, while RuntimeException 1 has been found by program versions other than Zest more rare and the number of times found is too small to come to any conclusion.

Only Zest has found RuntimeException 2 in 2 out of 20 runs and extension FAST once. Since the amount is too low, nothing can be said about the average time to find that failure.

Failure	Zest	FAST	QUAD	Entropic256	Entropic4096
Runtime Exception 1	9348s (9)	10621s (4)	10631s (3)	11269s (5)	13099s (5)
Runtime Exception 2	11697s (2)	9524s (1)	-	-	-
NullPointer Exception	27s (20)	23s (20)	27s (20)	30s (20)	22s (20)

Table 3: Average time to find each of the unique bugs found in CLOSURE

Rhino: Results of Rhi can be seen in the following table 4: As it can be seen in the table, some have been found at the similar times (IlligalStateException 1,2), some (VerifyErrorException) have been found by Zest, Entropic 256 and 4096 slighlty faster than FAST and QUAD, and some (NullpointerException) have been found by Zest, FAST and QUAD at the same time with Entropic finding it 1.5-2 times slower

Nothing can be concluded from ClassCastException 1, 2 and IlligalStateException 3, since the amount of times found is too low and it is probably due to the random nature of fuzzing algorithms.

Failure	Zest	FAST	QUAD	Entropic256	Entropic4096
ClassCast Exception 1	5427s (3)	10589s (3)	8907s (7)	6711s (3)	-
ClassCast Exception 2	11097s (4)	6627s (7)	11231s (8)	5271s (2)	6803s (5)
VerifyError Exception	1472s (20)	2845s (20)	2142s (20)	1356s (20)	1531s (20)
IllegalState Exception 1	64s (20)	58s (20)	64s (20)	55s (20)	46s (20)
IllegalState Exception 2	59s (20)	75s (20)	55s (20)	63s (20)	55s (20)
IllegalState Exception 3	-	-	-	13105s (1)	-
NullPointer Exception	431s (20)	506s (20)	459s (20)	653s (20)	808s (20)

Table 4: Average time to find each of the unique bugs found in RHINO

Overall, some of failures have been found faster by Zest, while others have been found faster by power scheduling algorithms. But AFLFast Power Schedules FAST and QUAD tend to find failures much slower than Zest, while Entropic with threshold 256 and 4096 tend to find the failures slightly slower. It could be the case, that the overhead for the algorithms slows down the process of finding unique failures.

Failures vs Unique Falures: As it can be seen in the tables 5 and 6, Zest produces on average one unique failure per 254.09 failures for Rhino, 1163.23 failures for Closure, 1.08 failures for BCE and 1 failure forAnt . FAST required 240.27 failures for Rhino, 1096.92 failures for Closure, 1.12 failures for BCE and 1. Quad required 310.39 failures for Rhino, 1214.35 failures for Closure, 1.074 failures for BCE and 1. Entropic256 required 191.83 failures for Rhino, failures for Closure, 819.64, 1.07 failures for BCE and 1. Entropic4096 required 186.68 failures for Rhino, failures for Closure, 765.56, 1.05 failures for BCE and 1.

While it does not necessary mean, that the lower amount of failures for each of the unique failures is better, it is in some cases better to generate less duplicate failures than to find failures.

Benchmark	Zest	FAST	QUAD	Entropic256	Entropic4096
Rhino	87	90	75	86	85
Closure	31	25	23	25	25
BCE	79	65	67	81	75
Ant	20	20	20	20	20

Table 5: Amount of unique failures found by each of the programs in each of the benchmarks over all 20 runs

Benchmark	Zest	FAST	QUAD	Entropic256	Entropic4096
Rhino	22106	21624	23279	16497	15868
Closure	36060	27423	27930	20491	19139
BCE	85	73	73	87	79
Ant	20	20	20	20	20

Table 6: Amount of total failures found by each of the programs in each of the benchmarks over all 20 runs

4.5 Summary

As it could be observed in Section 4, FAST and QUAD from AFLFast aswell as Entropic with both 256 and 4096 threshold produced less valid inputs, while producing more invalid inputs. Invalid inputs tend to require less time for execution and therefor more inputs have been executed. Since it is difficult to evaluate the overhead due to that reason, FAST and QUAD tend to produce around 70-100% of valid inputs at the same time, depending on the benchmark, while both Entropic versions produce around 30-80% of valid inputs. Also achieved coverage varies hardly on the benchmark. While in Rhino all the extensions seem to deliver roughly the same results, Entropic versions achieve from 90 up to 100% coverage from Zest baseline. FAST and QUAD lead towards same results except for Maven, where they only achieve 80% of Zest's coverage.

5 Threats to validity

While the experimental part of this work focuses on the differences between different approaches, it is important to mention, that this work, like any work with any evaluation of the algorithms, does not necessary makes conclusions about the algorithms themselves but about the implementations of those algorithms. While all the experiments have been run on a server without any access from other people or programs, all 5 variations of Zest, including baseline Zest, have been run simultaneously. That means, if one program had priority of the server's resources over another one, it could influence the work of other programs, which were running at the same time.

6 Conclusion

While fuzzing is one of the most successful vulnerability discovery techniques, it still has some aspects, which could be improved, e.g. reforming the cyclic queue of saved inputs with each element getting fuzzed the same amount of times into more efficient structure.

In this work, proposed in AFLFast and Entropic Power Scheduling algorithms have been implemented into generator-based fuzzer Zest and evaluated. While the result show, that the algorithms are not very suitable for Zest, as it is not only generator-based fuzzing algorithm but is also coverage-based, it would be interesting to further examine the functionality of Power Scheduling algorithms on other generator-based fuzzing tools or to further examine it via implementing and evaluating other Power Scheduling algorithms like [9].

7 Bibliography

References

[1] Afl vulnerability trophy case. `https://lcamtuf.coredump.cx/afl/`. Accessed: 2022-11-07.

[2] Jqf github repository. `https://github.com/rohanpadhye/JQF`. Accessed: 2023-05-30.

[3] A. Barkworth, L. Mcdonald, and M. Ijaz Ul Haq. Survey of software fuzzing techniques. 12 2021.

[4] M. Böhme. Stads: Software testing as species discovery. *ACM Trans. Softw. Eng. Methodol.*, 27(2), jun 2018.

[5] M. Böhme, V. J. M. Manès, and S. K. Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 678–689, New York, NY, USA, 2020. Association for Computing Machinery.

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[8] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[9] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 533–544, New York, NY, USA, 2019. Association for Computing Machinery.

[10] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[11] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2021.

[12] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den June 2, 2023