SSG-AFL: Vulnerability detection for Reactive Systems using Static Seed Generator based AFL

Sangharatna Godboley
Department of CSE
National Institute of
Technology Warangal, India
sanghu@nitw.ac.in

Arpita Dutta
School of Computing
National University of Singapore
Singapore
arpita@comp.nus.edu.sg

P. Radha Krishna
Department of CSE
National Institute of
Technology Warangal, India
prkrishna@nitw.ac.in

Durga Prasad Mohapatra
Department of CSE
National Institute of
Technology Rourkela, India
durga@nitrkl.ac.in

***Abstract*—Fuzzing is a popular and highly effective technique for software testing especially vulnerability detection. Fuzzing includes the random mutation of well-formed program inputs using dynamic program analysis. Though fuzzing is an active area of research, less systematic efforts have investigated to understand as well as to generate powerful input seeds for a fuzzer. Reactive systems are used in different applications such as web services, decision support systems, and logical controllers. These systems are quite complex and bigger, hence the validation process becomes tedious. In this work, we propose a static seed generator that helps to accelerate the performance of existing fuzzers. In this paper, we validate the reactive systems using our approach by detecting vulnerability. To evaluate the performance of our developed seeder, we experimented with 100 Rigorous Examination of Reactive Systems (RERS) C-programs. Experimental results show that our approach SSG-AFL is superior as compared to the AFL with random seeds. SSG-AFL shows 59.75% winning programs after running all four phases as compared to Random-AFL.**

***Index Terms*—Security, Vulnerability Detection, Fuzzing, Reactive System**

I. Introduction

In recent time, software vulnerabilities are one of the root cause for cyberspace security threats. As per RFC 2828 [11], vulnerability is a defect that create disturbance in the perfect execution of a system and violate the system security. The vulnerabilities can be introduced at the design, operation, implementation or system maintenance [4], [5], [28].

By considering the severity of this situation, several testing techniques are deployed for the early detection of the software vulnerabilities [8], [14]. Among those techniques, fuzzing is the most popular one [9], [25], [26]. Fuzzing is a random test generation technique. It was first instigated to discover the Unix utility bugs in the early 1990s [11]. From that time onwards, fuzzing became one of the most scalable and effective testing method to detect the crashes and vulnerabilities in the commercial off-the-shelf (COTS) software systems. Fuzzing is widely used in the mainstream software industries Google [6], Microsoft [15], and Adobe [1] to ensure the quality of their developed software products.

Fuzzer generates a large amount of test inputs with the hope of target the unintended program behaviors and discovers the crashes and faults. Initial seeds are important to populate the quality test cases. This ultimately decides the overall effectiveness and efficiency of the fuzzer. However, till this time, very less attention is given to the seed selection strategies. There are few mutation based [3], [17] and generation based [16] seed selection techniques are available but these techniques are not applicable to different software systems [23]. Also, there is no consensus on which seed selection approach is the best. Even though, we know that the fuzzing outcomes are completely seed dependent.

To resolve the above highlighted issues, we propose a static seed generation technique in this paper. In this technique, we select all the constant values attached with the input variables as initial seeds for effective fuzzing. Also, to not miss out any boundary conditions, we consider one greater as well as one lower value of each seed value in the initial set. However, this set may contains a few seeds which lead to the same path. Therefore, to mitigate this issue, we use *afl-cmin* and *afl-tmin*, two AFL-based seed optimizers, to generate optimized seeds. We evaluate the effectiveness of our proposed static seed generator over 100 C-programs using AFL [27]. Further, we compare its efficiency and fault detection capability with random seed generation.

Rest of the article is organized as follows. We survey the related work in Section II. Subsequently, we discuss our proposed approach in Section III. Followed by this, we explain our approach with a working example in Section IV. Our experimental results are discussed in Section V. Finally, we conclude in Section VI.

II. Related Literature

In this section, we present important related studies with our work.

Herrara et al. [7] reported a systematic investigation over six different seed selection strategies used in fuzzing to detect bugs and vulnerabilities. They have evaluated these techniques over three large scale corpus minimization tools. Their finding shows that the fuzzing outcomes are vary significantly based on the initially selected seeds. Because the initial seeds are the main responsible for the bootstrapping of a fuzzer. They have also suggested to make a careful selection of seed files and to never evaluate fuzzer with a single seed file.

Lyu et al. [13] developed a smart seed generation called SmartSeeder for efficient fuzzing. It is a generic system to generate effective seeds. SmartSeeder is based on unsupervised machine learning model to learn and create high value binary seeds. Authors have evaluated SmartSeeder with American Fuzz Lop (AFL) over 12 open-source applications with different input formats like .bmp, .mp3, and .flv. Their empirical analysis shows that SmartSeeder is able to discover 5040 extra unique paths, twice unique crashes and 16 new vulnerabilities as compared to the existing best seed selection strategy. They have also verified the compatibility of SmartSeeder with different fuzzing tools and found it effectively compatible.

Wang et al. [23] proposed a data driven based seed generator approach for fuzzing called Skyfire. Their technique borrows information from a vast set of existing samples to create useful seeds. It takes a grammar and a corpus as input and process them in two steps to generate well-distributed input seeds. Skyfire first learns the probabilistic context sensitive grammar (PCSG) to specify both semantic rules and syntax features. In the second step, it used the learned PCSG rules to the seed inputs. They experimented their seed generator with AFL to fuzz and test the open-source XML and XSLT engines for libXML2, libXSLT, and Sablotron etc. Their empirical results show that Skyfire has effectively generated well-distributed seeds which able to improve the function coverage by 15% and line coverage by 20% on an average. It also improves the bug-finding capability. Skyfire exposed 32 denial-of-service bugs and 19 new memory corruptions bugs in the rendering engine of internet explorer-II which is also a closed Java project.

Liang et al. [12] proposed an improved grey-box fuzzing technique called DeepFuzzer. It is based on qualified seed generation, balanced seed selection and hybrid mutation of seeds. They have used symbolic execution [10] to generate the qualified initial seeds. Subsequently, applied a statistical seed selection strategy to maintain a balance between frequency of seeds. Further, they have used a hybridization of restricted and random mutation strategies on the selected seeds to maintain a dynamic balance between deep search and global exploration. They have evaluated DeepFuzzer over Google fuzzer-test-suite, which is a famous and widely used benchmark consisting of real-world programs. Empirical results showed that DeepFuzzer discovers 30%, 40% and 35% more unique crashes, program paths and branches than AFL respectively.

Wang et al. [24] proposed a deep learning models based high quality seed generation technique called LAFuzz. It is an offline combination of generation based and mutation based fuzzing techniques. empirical results shows that their proposed LAFuzz-Attention, and LAFuzz-LSTM outperforms AFL in terms of both code coverage and crash discovery. LAFuzz-Attention performs 7.67% and 82.39% more effectively than AFL in terms of code coverage and unique crash detection respectively. Similarly, LAFuzz-LSTM performs 7.55% and 30.19% more effectively than AFL in code coverage and unique crash detection respectively.

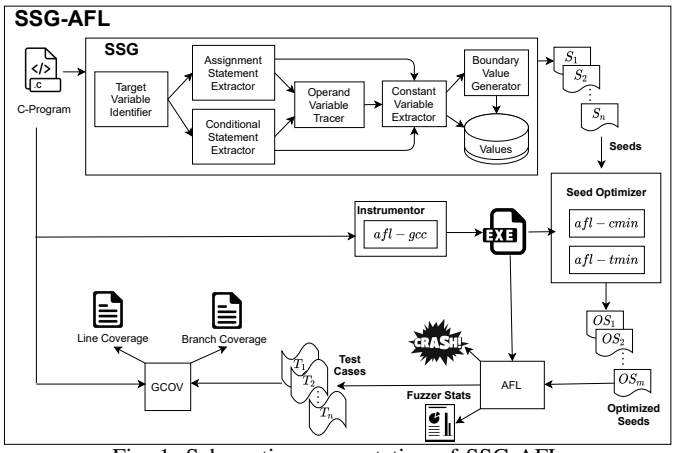


Fig. 1: Schematic representation of SSG-AFL

III. Proposed Work: SSG-AFL

In this section, we discuss the framework and detail description of our proposed approach.

A. Framework

Fig. 1 shows the schematic representation of our proposed work Static Seed Generator based AFL (SSG-AFL). There are five components in SSG-AFL as shown in Fig. 1. These are 1. Static Seed Generator (SSG), 2. Instrumentor, 3. Seed Optimizer, 4. AFL, and 5. GCOV. The flow starts with supplying *C-Program* into SSG to produce *Seeds*. Also, the same *C-Program* supplied into Instrumentor to produce instrumented (executable) program, that is, *EXE*. Now, these *Seeds* and *EXE* imparted into Seed Optimizer component. Then Seed Optimizer produces *Optimized Seeds* which supplied into AFL along with *EXE* to produce *CRASH Report*, *Fuzzer Stats*, and *Test Cases*. Finaly, the *C-Program* and *Test Cases* supplied into GCOV (coverage analyzer comes with GNU) to produce *Line Coverage* and *Branch Coverage* information.

B. Details

In this section we provide the details of the components.

a) Static Seed Selector (SSG): SSG is a proposed and implemented component. It takes a *C-Program* and generates *Seeds*. As name of the component highlights that the mechanism used is static in nature. SSG contains six sub-components. These are *Target Variable Identifier*, *Assignment Statement Extractor*, *Conditional Statement Extractor*, *Operand Variable Tracer*, *Constant Variable Extractor* and *Boundary Value Generator*.

Target Variable Identifier identifies the *scanf()* or *read()* or *fget()* etc. statements present in the supplied C-Program, and collect all the *variables*. Note that any test case generator requires the non-concretized variables, which means variable's value to be supplied by user or a tool. Now, these collected variables will be used to extract *Assignment Statements* and *Conditional Statements* present in the program. For an *Assignment Statement*, the targeted variable is at LHS, and the constant is present in RHS of assignment operator "=", in

this case *Constant Variable Extractor* extracts that constant and store it for using as seed. Also, if there is a *Conditional Statement* for which the targeted variable is present in either of the operands, and the other operand is a constant, then this value will be extracted by *Constant Variable Extractor* to use it as a seed. It is possible that there may be some assignment statements, arithmetic statements, and conditional statements in the program for those targeted variables exist without only constants for examples, $a = b$, $p = q + 1$, and $x > y$. In such case, *Operand Variable Tracer* will track the information of the variables present in the operand until they provide the constant values. It is a backtrack approach to trace the constant values from the variables, so that those values can be utilise as seeds. Finally, all these extracted constants stored to use as seeds as well as these constants are forwarded into *Boundary Value Generator*. This component takes the constants and generates the boundary values with simple ± 1 operations.

b) Instrumentor: It takes *C-Program* and produces instrumented version *EXE*. It uses *afl-gcc* [1]. This component is used instead of *gcc*. It is used to compile the code with the required targets and instrumentation for AFL (afl-fuzz).

c) Seed Optimizer: This takes *EXE* and *Seeds* to produce *Optimized Seeds*. This component has two sub-components viz. *afl-cmin*[2] and *afl-tmin*[3]. Both the components optimise the seeds/test inputs.

The component *afl-cmin* is based on a greedy distillation algorithm, and it has a unique technique to coverage. This discards AFL's notion of edge coverage to categorize initial seeds at starting time. Here, AFL counts approximate edge frequency, not just whether the edge has been taken. During the optimisation it chooses the smallest seed in the collection corpus that covers a given edge, and then performs a greedy distillation technique.

The *afl-tmin* algorithm uses a more rigorous, iterative algorithm, and also attempts to perform alphabet normalization on the trimmed files. Initially it selects the operating mode. In case the initial input crashes the target binary, it will run in non-instrumented mode and keep tweaks that produce a small file, but still crash the target. In case that target is non-crashing, it uses an instrumented mode and keeps only the tweaks that produce exactly the same execution path.

d) AFL: This takes *EXE* and *Optimized Seeds* to produce *CRASH Report*, *Test Cases*, and *Fuzzer Stats*. The main code of AFL is *afl-fuzz*[4]. The *afl-fuzz* is highly deterministic, and progresses to random stacked modifications. It does test case splicing only at a later stage. The *afl-fuzz* uses Sequential bit flips with varying lengths and stepovers, Sequential addition and subtraction of small integers, and Sequential insertion of known interesting integers (0, 1, etc), The idea is based on the tendency to generate compact test cases and small diffs between the non-crashing and crashing inputs. Also, the non-deterministic steps include stacked bit flips, insertions, deletions, arithmetics, and splicing of different test cases. The AFL is a well known fuzzer and more technical details can be found [5].

e) GCOV: This is a source code coverage analysis tool comes with GCC. It is used to show the covered and uncovered parts for the programs after running Test Cases. GCOV [6] can be used as a profiling tool. This helps analyze the program performance. This gives basic performance statistics such as Lines Covered and Branches Covered.

IV. Working Example

In this section, we show the execution of a working example. To compare the results we have considered Random Seed for AFL and named it as Rand-AFL in this paper (interchangeably we call it as Mode1). Listing 1 shows an artificial reduced version of one RERS program. This program has two functions viz. *main* and *calculate_output*. Since the originally program loop was infinite bound which cannot be run, hence we have given a bound of 1000. There is an artificial bug seeded at line number 25 "*assert(0)*". Expectation is that this bug will be detected by any of the modes.

Listing 2 shows the random seed generated for Rand-AFL. Listing 3 shows the test cases generated by Rand-AFl. The value 46 for *input* variable is in Queue folder however a meaningful value i.e. 55 for variable *input* got created, which finds the target "*assert(0)*". We can observe that the predicate at line number 21 in Listing 1 "$if((b == 35\&\&((input == 55)\&\&(a == 11\&\&z == 1))))$ " has an atomic condition "$input == 55$", which is essentially to be true to find the target "*assert(0)*". Here, Rand-AFl found the target from the seed provided "input=42", but it took some time to generate a meaningful test input i.e. "input=55". Listing 4 shows the time analysis for the process by Rand-AFL. It took **5.17 seconds** to detect the bug. The coverage information (Line Coverage **42.11%** and Branch Coverage **38.89%**) for Rand-AFl is reported in Listing 5.

Now, we explain about SSG-AFL (interchangeably we call it as Mode2). Here, there is only one *scanf-statement* in the program. We extract the variable i.e. "*input*" and call it as a target variable. The component SSG statically generates total **21 seeds** as shown in Listing 6. These seeds supplied into *afl-cmin* and it optimised the seeds into only **2 seeds** as shown in Listing 7. Further, we supply these seeds into *afl-tmin* that gives only **1 seed** as shown in Listing 8. We consider this seed as the more meaningful and carefully designed seed as compared the random seed selection process, because we collect the information from the program itself, whereas the random seed takes the value from linux process id which is outside of the program. Listing 9 shows the test inputs generated by SSG-AFL. The Queue test cases are "input=5,\B5". Note that the value '\B5' is the input prepared after flipping byte as a strategy and raw in nature. During the cleaning to make it readable format it can be used to compute

[1]https://github.com/google/AFL/blob/master/afl-gcc.c
[2]https://github.com/mirrorer/afl/blob/master/afl-cmin
[3]https://github.com/google/AFL/blob/master/afl-tmin.c
[4]https://github.com/google/AFL/blob/master/afl-fuzz.c
[5]https://github.com/google/AFL/blob/master/docs/technical_details.txt
[6]https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro

coverage. But, our main value here is "$input = 5$" which gets fuzzed to "$input = 55$" faster or quickly as compared to random. Listing 10 shows the time analysis for SSG-AFL. It took **4.63 seconds** to detect the bug. The coverage information (Line Coverage **75.00%** and Branch Coverage **72.22%**) for SSG-AFl is reported in Listing 11. We can see that SSG-AFL is superior as compared to Rand-AFl, and it is able to generate meaningful seeds and hence detecting the bug early.

From the fuzzer statistics from Rand-AFL and SSG-AFL, it has been observed that Rand-AFl exercised a total of **1177 executions done**, whereas SSG-AFL exercised a total of **201 executions done**. It shows that SSG-AFL requires less efforts to detect a bug. Regarding the total paths, so both the modes have covered **2 paths**.

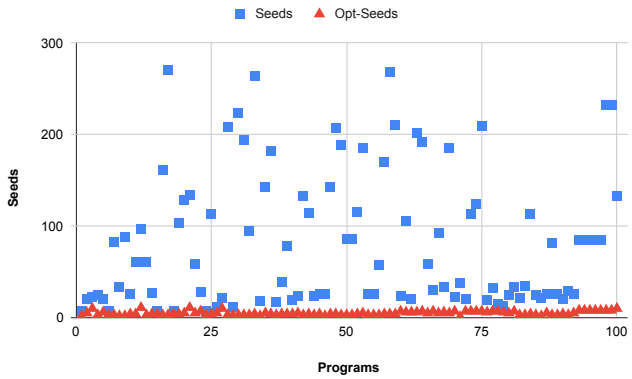


Fig. 2: Seeds generated

Listing 1: A C-program

```
#include <stdio.h>
#include <assert.h>
#define BOUND 1000
void calculate_output(int);
int z = 1, a = 11, b = 35, c = 35, d = 2;
 void calculate_output(int input) {
      z = 1;
   if((b == 33 && z==1 )) {
     if((d == 2 && z==1 )) { }
     if(( z==1 && d == 5) && (((input == 2) &&
         ( z==1 && b == 33)) && d == 5)) {
     z = 0; d = 2; } }
   if((b == 35 && z==1 )) {
     if((a == 11 && z==1 )) {
     if((b == 35 && ((input == 55) && (a == 11
         && z==1 )))) {
     z= 0; b = 32 ; c = 32 ;
     assert(0);}
          if((a == 11 && (((input == 5) &&
              z==1 ) && b == 35))) {
     z = 0; b = 33; d = 5; }}}}
int input;
int main(){
  for (int FLAG=0;FLAG<BOUND;FLAG++){
   scanf("%d", &input);
   calculate_output(input);}
return 0; }
```

Listing 2: Seed for Rand-AFL

```
input={42}
```

Listing 3: Test Inputs for Rand-AFL

```
--Queue Test Input(s)----
input={46}
-------------------------
--Crash Test Input(s)----
input={55}
```

Listing 4: Time analysis for Rand-AFL

```
**Total runtime in seconds 5.178477
```

Listing 5: Coverage for Rand-AFL

```
Lines executed:54.17% of 24
Branches executed:38.89% of 36
```

Listing 6: Static Seed(s) for SSG-AFL

```
input={1,2,0,5,6,4,3,33,34,32,35,36,55,56,54,
11,12,10,9,8,7}
```

Listing 7: Optimised Seed(s) using afl-cmin for SSG-AFL

```
input={5,3}
```

Listing 8: Optimised Seed(s) using afl-tmin for SSG-AFL

```
input={5}
```

Listing 9: Test Inputs for SSG-AFL

```
--Queue Test Input(s)----
input={5,'\B5'}
-------------------------
--Crash Test Input(s)----
input={55}
```

Listing 10: Time analysis for SSG-AFL

```
**Total runtime in seconds 4.631974
```

Listing 11: Coverage for SSG-AFL

```
Lines executed:75.00% of 24
Branches executed:72.22% of 36
```

V. Experimental Study

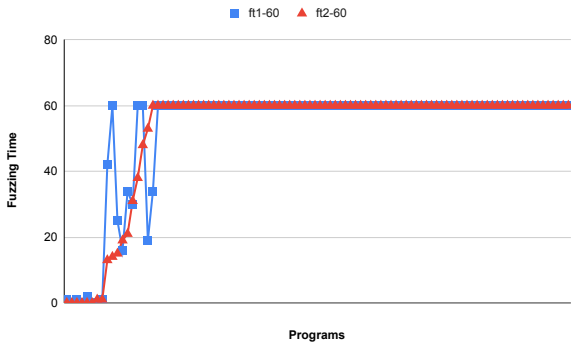
In this section, we discuss the setup, benchmarks tested, results evaluation and discussion on results.

A. The Set Up

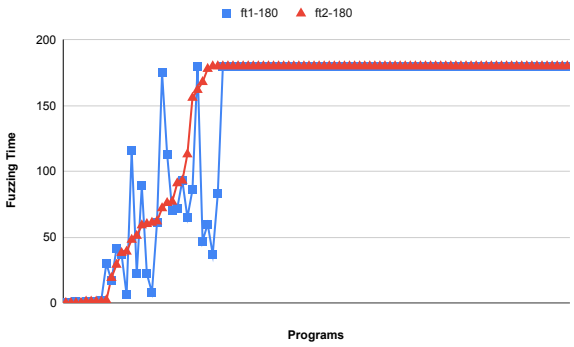
We used an Intel Core i7-9700 CPU @ 3.00GHz × 8 Linux box (64-bit Ubuntu 16.04) with 64 GB RAM. All the input programs considered for our study are written in ANSI-C format. For result comparison, we consider *AFl* with RANDOM seed as our baseline because it is a state-of-the-art tool. The programs and all the raw experimental details are provided in the supplementary artifacts [2].

B. Benchmarks Tested

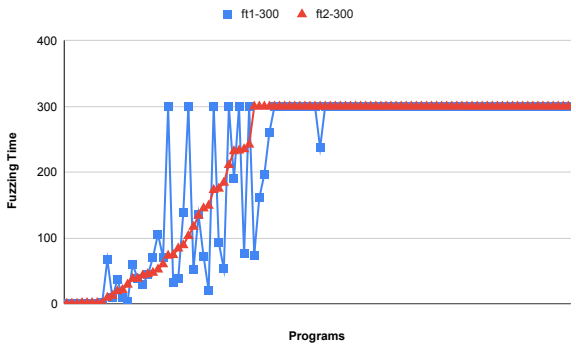
Reactive systems appear everywhere, e.g. as Web services, decision support systems, or logical controllers. The testing techniques are as diverse due to their complex structure. RERS programs are automatically synthesized to exhibit chosen properties, and then enhanced to include dedicated dimensions of difficulty, ranging from conceptual complexity of the properties such as reachability, full safety, liveness etc. over size of the reactive systems (a few hundred lines to millions



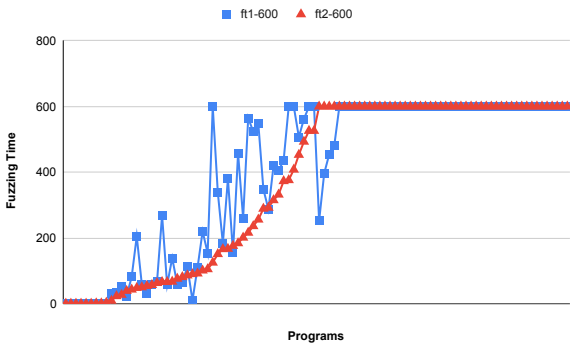
(a) ft1-60 (Rand-AFl) vs. ft2-60 (SSG-AFL)



(b) ft1-180 (Rand-AFl) vs. ft2-180 (SSG-AFL)



(c) ft1-300 (Rand-AFl) vs. ft2-300 (SSG-AFL)



(d) ft1-600 (Rand-AFl) vs. ft2-600 (SSG-AFL)

Fig. 3: Charts show the Fuzzing Times

of them), to exploited language features (arrays, arithmetic at index pointer, and parallel message passing). Hence, we assume that we are considering RERS programs that replicate the real-world applications from Avionics, Banking, Medical and Railways etc. [7]

In total, we have tested 100 programs taken from RERS [21]. They are from RERS challenge competition in years 2017 [18]–[20], and 2018 [22]. These programs are from the small and moderate size group and easy to hard categories. Though programs are originally unbounded, we set 1000 bounds considering as an infinite loop (to bound the program). This is because the unbounded programs mean the programs have infinite loop bound without any exit criteria.

C. Discussion on Results

In this section we discuss on the results in detail.

Fig. 2 shows the total number of seeds generated for 100 programs. The Blue Squares are dominating the Red triangles in Fig. 2. Which shows that even though SSG produces a lot of values for target variables but Seed Optimiser shapes them into meaningful seeds.

Fig. 3 shows the fuzzing times for Rand-AFL and SSG-AFL on four phases. The Sub-figures 3a, 3b, 3c, and 3d show the fuzzing times charts for 60, 180, 300, and 600 seconds timeouts phases. Horizontal lines for Blue squares and Red lines for programs show the timeout for both the modes. Except few programs, Lines with Red triangles are most of the time below the Line with Blue squares. This shows that SSG-AFl is faster and having less timeouts.

Now, we will discuss on the winning and loosing cases of the program for all the phases. First, we divide the program in Four groups:

1) **Group 1 (G1)**: Rand-AFL and SSG-AFl terminate[8] within timeout.
2) **Group 2 (G2)**: SSG-AFl terminates within timeout but Rand-AFL could not.
3) **Group 3 (G3)**: Rand-AFL terminates within timeout but SSG-AFL could not.
4) **Group 4 (G4)**: Neither Rand-AFL nor SSG-AFL terminate within timeout.

Table I shows the programs in groups with winning cases. Table Ia shows four groups. We can observe that Column G1 gets increase as the timeout is increasing. On the other hand in Column G4 the number of programs get decrease. It is inversely proportion to each other. Table Ib shows the winning programs for Group 1. There is a total of 14, 29, 31 and 47 programs in 60, 180, 300, and 600 phases respectively. But, out of these programs SSG-AFl has 10, 15, 14, and 36 which are better than Rand-AFL. Similarly, Table Ic shows the

[7]This information is taken from the main website of The RERS Challange http://rers-challenge.org/

[8]Note that terminate means bug has been detected and fuzzing has been stopped.

winning programs for Group 2. There is a total of 3, 1, 6 and 5 programs in 60, 180, 300, and 600 phases respectively. It is to be noted that here Rand-AFL has 0 program in winning cases. So it is a clear case of good results. Also, Table Id shows the winning programs for Group 3, where Rand-AFL is winning over SSG-AFL. The Table Ie shows winning programs for Group 4. There is a total of 82, 69, 58, and 45 programs in 60, 180, 300, and 600 phases respectively. Out of these 63, 56, 52, and 68 programs have same or more number of total paths.

Finally, overall summary can be observed from Table If. In total for phase 1 (60 sec) SSG-AFL has **63** winning programs out of 100 programs and Rand-AFL has **37**. For phase 2 (180 sec) SSG-AFL has **56** winning programs out of 100 programs and Rand-AFL has **44**. For, phase 3 (300 sec) SSG-AFL has **52** winning programs out of 100 programs and Rand-AFL has **48**. Finally, for phase 4 SSG-AFL has **68** winning programs out of 100 programs and Rand-AFL has **32**, it shows the benefits of SSG-AFL.

TABLE I: Programs in groups with winning cases (Green cells)

(a) Four Groups

	G1	G2	G3	G4
60	14	3	1	82
180	29	1	1	69
300	31	6	5	58
600	47	5	3	45

(b) Group 1

	Rand-AFL	SSG-AFL
60	4	10
180	14	15
300	17	14
600	11	36

(c) Group 2

	Rand-AFL	SSG-AFL
60	0	3
180	0	1
300	0	6
600	0	5

(d) Group 3

	Rand-AFL	SSG-AFL
60	1	0
180	1	0
300	5	0
600	3	0

(e) Group 4

	Rand-AFL	SSG-AFL
60	32	50
180	29	40
300	26	32
600	18	27

(f) Overall Summary

	Rand-AFL	SSG-AFL
60	37	63
180	44	56
300	48	52
600	32	68

VI. Conclusions

In this paper, we propose a novel approach called Static Seed Selector based AFL (SSG-AFL) for vulnerability detection. We considered AFL with random seeds (Rand-AFL) as our baseline. SSG-AFL was static in nature and it is the big advantage because SSG consumed few milliseconds to generate good number of seeds. We have discussed about the detailed design of the SSG-AFL. Also, we have discussed the approach with a working example. Finally, we have rigorously discussed the results and showed the winning and losing programs. Our proposed approach SSG-AFL has overall 63%, 56%, 52% and 68% winning program for phases 1 to 4 respectively in contrast to Rand-AFL. On an average for four phases and 100 program SSG-AFL has **59.75%** winning programs.

In future we will work to improvise SSG and Seed Optimiser. We have observed that afl-cmin and afl-tmin have optimised the set of seeds a lot which have removed some important seeds which could have helped to detect the bug earlier.

References

[1] Adobe reader and acrobat security initiative. http://blogs.adobe.com/security/2009/05/adobereaderandacrobatsecur.html, 2009.

[2] Raw experimnetal data, 2021.

[3] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *ISSTA*, pages 12–22, 2011.

[4] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. Meuzz: Smart seed scheduling for hybrid fuzzing. In *23rd RAID*, pages 77–92, 2020.

[5] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEESSP*, pages 679–696. IEEE, 2018.

[6] Google online security blog – fuzzing at scale. https://security.googleblog.com/2011/08/fuzzing-at-scale.html, 2011.

[7] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In *30th ISSTA*, pages 230–243, 2021.

[8] Paul C Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2013.

[9] SungJin Kim, Jaeik Cho, Changhoon Lee, and Taeshik Shon. Smart seed selection-based effective black box fuzzing for iiot protocol. *Journal of Supercomputing*, 76(12), 2020.

[10] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[12] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[13] Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, and Jing Chen. Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606*, 2018.

[14] Rajib Mall. *Fundamentals of software engineering*. PHI Learning Pvt. Ltd., 2018.

[15] Sdl process: Verification. https://www.microsoft.com/en-us/sdl/process/verification.aspx.

[16] Peach fuzzer platform. http://www.peachfuzzer.com/products/peach-platform/.

[17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[18] Rigorous examination of reactive systems (rers-2017): Sequential ltl problems, 2017.

[19] Rigorous examination of reactive systems (rers-2017): Sequential reachability problems, 2017.

[20] Rigorous examination of reactive systems (rers-2017): Sequential training problems for rers 2017, 2017.

[21] RERS:, June 2018.

[22] Rigorous examination of reactive systems (rers-2018): Sequential training problems for rers 2018, 2018.

[23] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEESSP*, pages 579–594. IEEE, 2017.

[24] Xiajing Wang, Changzhen Hu, Rui Ma, Binbin Li, and Xuefei Wang. Lafuzz: neural network for efficient fuzzing. In *ICTAI*, pages 603–611. IEEE, 2020.

[25] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, 7:36340–36352, 2019.

[26] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. Pathafl: Path-coverage assisted fuzzing. In *Asia CCS*, pages 598–609, 2020.

[27] M Zalewski. Afl—american fuzzy lop, 2015.

[28] Yiru Zhao, Ruiheng Shi, Lei Zhao, and Yueqiang Cheng. Alphafuzz: Evolutionary mutation-based fuzzing as monte carlo tree search. *arXiv preprint arXiv:2101.00612*, 2021.