AV-AFL: A Vulnerability Detection Fuzzing Approach by Proving Non-reachable Vulnerabilities using Sound Static Analyser

Sangharatna Godboley[a], Kanika Gupta and G. Monika Rani[b]
Department of CSE, NIT Warangal, Telangana, India

Keywords: Fuzzing, Static Analyzer, Vulnerability Detection.

Abstract: The correctness of software depends on how well the vulnerabilities of the program are detected before the actual release of the software. Fuzzing is an effective method for vulnerability detection but it also comes with its drawback. The traditional fuzzing tools are less efficient in terms of speed and code coverage. In this paper, we demonstrate how a fuzzer works more efficiently when the input to it is given based on static analysis of the source code. We introduce the Alarmed Vulnerabilities-based American Fuzzy Lop (AV-AFL) tool that eliminates the unreachable targets from the program by analyzing the source code using the FRAMA-C tool (a sound static analyzer). The method uses Evolved Value Analysis (EVA) plugged-in with FRAMA-C tool to report alarms of possible run-time errors and gives the improvised program as an input to the AFL fuzzer. Experimental results show that the AV-AFL produces better results in total **71.11% of 45** programs than AFL in terms of vulnerability detection.

1 INTRODUCTION

Software Testing is the process of evaluating the usability or capability of a program. It is a process where maximum errors need to be found as the execution of a program proceeds which aims at getting zero-defect software. Software testing is an important phase in software development for assessing the quality (Godboley et al., 2015; Godboley et al., 2017a; Godboley et al., 2016; Godboley et al., 2021; Godboley et al., 2017b; Godboley et al., 2018a; Godboley et al., 2018b). Software testing involves verification, validation, and error detection. These techniques are involved so that software can be made more secure and problem-less.

Research has been done to overcome these vulnerabilities by early detection(Iorga et al., 2020). There can be various types of analysis made on the software to detect the vulnerabilities, like static analysis, dynamic analysis, fuzzing etc. Fuzzing is a better technique than most of the methods because it is easy to use and execute, highly scalable, and can work in the absence of source code (provided with the executable code).

The traditional fuzzers are incapable of finding the unreachable targets[1] and, as a result AFL attempts to explore all possible edges, which requires non-essential execution time. On the other hand, the static code analysis tools are useful in knowing the behavior of the source code without executing the program. The static analysis gives the potential vulnerabilities in the program but fails to infer concrete test cases triggering those bugs.

Thus, in this paper we present a method Alarmed Vulnerability based American Fuzzy Lop (AV-AFL), that leverages the static analysis of the code to produce an efficient result output from the fuzzer. The tool first obtains the locations which are unreachable and eliminates them from the original source code, the new and improvised source code is supplied to the fuzzer which uses random and minimized seed generation to generate the seeds to trigger the vulnerabilities at the point of concerns. In this way, unnecessary attempts of edge exploration are avoided.

We use Frama-c (potassium)[2] for static analysis of the code. Frama-c is a sound static analyzer tool that helps in reporting the possible runtime errors in form of alarms. The value analysis (EVA plug-in) in the Frama-c (Baudin et al., 2021) to report alarms of pos-

[a] https://orcid.org/0000-0002-6169-6334
[b] https://orcid.org/0000-0002-1662-5764

[1] Vulnerabilities and Targets have been interchangeably used in this paper.

[2] https://frama-c.com/html/installations/potassium.html

Godboley, S., Gupta, K. and G. Monika, .
AV-AFL: A Vulnerability Detection Fuzzing Approach by Proving Non-reachable Vulnerabilities using Sound Static Analyser.
DOI: 10.5220/0011032900003176
In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 301-308
ISBN: 978-989-758-568-5; ISSN: 2184-4895

sible run-time errors thereby deducing the unreachable errors for the program. The new improvised input is generated based on the Frama-c output, this is supplied as an input to the AFL[3] along with the random seed generation mechanism. The program proposed uses the minimized corpora for vulnerability detection. In other words, we reduce the searching tasks for the fuzzer by eliminating unreachable vulnerabilities from the code as a result we expose more reachable vulnerabilities. Finally, we report the status of each vulnerability whether *Known* or *Unknown*. We have observed that AV-AFL could report a good number of Known targets as compared to AFL.

Rest of the article is organized as follows. We explain a few important basic concepts to understand our work in Section 2. We survey the related work in Section 3. Subsequently, we discuss our proposed approach in Section 4. Followed by this we show our experimental results in Section 5. Finally, we conclude in Section 6.

2 BASIC CONCEPTS

In this section, we explain some important basic concepts which are required to understand this paper.

Definition 2.1 (Fuzzing)**.** Fuzzing is one of the software testing techniques which is used to discover errors and security loopholes in the software code. Fuzzer proceeds by taking input value known as seed, mutates it to generate inputs and analyze the program for crashes due to them.

Definition 2.2 (Static Analysis)**.** In this type of code analysis, the source code is required and the code is not executed. But, it is interpreted using formal means to get valuable information regarding the source code. The Frama-C code is used in this project to carry out the static analysis.

Definition 2.3 (Targets)**.** The bugs in the source code which can be a threat to the security of the system and can be used by the hackers for malicious use are known as Targets. For example, div by zero is a target that can be exploited.

Definition 2.4 (Reachable Targets)**.** Reachable targets are the bugs in the source code which are reachable. These bugs are threat to the security of the program since they can lead to any crashes. To avoid exploitation from attackers using these vulnerabilities, the detection of such targets is important.

Definition 2.5 (Unreachable Targets)**.** Unreachable targets are the bugs in the source code which are not reachable. These bugs are not a threat to the security of the program but searching and its unknown status leads to loss of searching time. These targets are not useful during the fuzzing process since they cannot lead to any crashes. Elimination of Unreachable targets from source code prioritize the searching of remaining targets.

Definition 2.6 (Alarmed Targets)**.** Alarmed targets are the bugs in the source code which can be reachable or unreachable. The alarms are reported by a sound static analyzer which confirms that the targets which have not been alarmed are surely unreachable.

Definition 2.7 (Unknown Targets)**.** Unknown targets are the bugs in the source code which are neither proved as reachable or unreachable. After using sound static analyzer and fuzzer till time budget, the targets whose state remains unspecified are called as Unknown targets.

3 RELATED WORK

There can be various types of analysis made on the software to detect the vulnerabilities, like static analysis, dynamic analysis, fuzzing, etc.

Flinder-SCA (Kiss et al., 2015) is a method that is a combined verification tool for the detection of security bugs. The paper presents the application of the tool for the OpenSSL/HeartBeat Heartbleed vulnerability which is noticed in the network when there is a check done to note whether the server is up and running or not and is able to encipher the incoming packets using SSL techniques. It is a combination of static analysis and dynamic analysis. It works in different phases, the first phase uses Frama-c to reports the alarms using value, then the program is sliced. This process is taint analysis. The final step is the fuzzing which proves the potential alarms.

CURSOR (Signoles, 2021) is a method that uses static analysis and runtime checking of the assertions in order to make the source code attack resistant in an efficient way. It produces counter measure that are used at runtime to strengthen the program. The tool is tested by using the Frama-C framework, the results are shown based on real-life example of Apache webserver. CURSOR is proved to present a counter for some attacks based on Common Weakness Enumeration (CWE) entries. CURSOR identifies the functionalities that are to be analyzed to make the program stronger. Then, the Frama-c plugin value is used to get alarms. An intermediate code with CWE alarms is then produced and then after instrumentation the code is made CWE proof.

[3]https://github.com/google/AFL

BugMiner (Rustamov, 2021) is a target-oriented hybrid fuzzing that combines the fuzzing and dynamic symbolic execution, which is used to enhance the directed fuzzing process. It aims to mutate the input quickly and cover the path that consists of hard-to-reach vulnerabilities.

Orthrus (Shastry, 2017) a tool that make use of smart fuzzing by taking the information from the static analysis of the source code. It proves to be more effective than the modern fuzzers. It pre-processes the input to the fuzzer to increase the quality of the result. The output of the static analysis is the input dictionaries which is derived by the static analysis of the program. Based on this dictionary the fuzzer is guided to increase the test coverage. The results are produced by the experiment on network application nDPI and tcpdump.

AFLGo (Böhme et al., 2017) It is a Directed gray-box fuzzing tool. In this, the input generation is done so that they are able to reach a given set of target locations. Therefore there is a need for deterministic strategies for specified vulnerabilities. AFLGo is experimentally proven to outperform directed symbolic execution fuzzing and traditional gray-box fuzzing. However, Directed gray-box fuzzing is useful only when bugs are defined and not when they are unknown.

All the above work proves to be beneficial for the security of the program. The previous works do not take into consideration the ineffective fuzzing that is done while executing and testing the unreachable targets. Thus, this paper focuses on those vulnerabilities which unnecessarily increase the execution time of fuzzer and delay the crash detection. The method proposed in this system effectively detects crashes by only considering the sensible vulnerabilities i.e. the alarmed vulnerabilities.

4 PROPOSED APPROACH

In this section, we discuss our proposed approach in detail. First, we discuss the overall framework of AV-AFL in details, algorithmic description, followed by one working example.

4.1 Framework

Fig. 1 shows the framework of *AV-AFL*. The proposed approach AV-AFL is the integration of the static analysis of the source code along with the fuzzing methodology to detect vulnerabilities efficiently. The flow of the model begins with the original *C-Program* being supplied to the sound static analyser i.e. *Frama-C* (Baudin et al., 2021) which uses the *EVA* plug-in to get the detail of *Alarms* from the original *C-Program*. For Frama-C all the targets were annotated as "div-by-zero" errors. Later for fuzz, we replace the error with assert(0). The locations of alarmed vulnerabilities are used to target for Fuzzing. It means the non-alarmed targets from C-Program have been proved as Unreachable targets. So, the component *Code Refiner* takes *C-Program* and List of *Alarms* and produces *Refined C-Program*. This is an improvised version of the original program since it contains only meaningful targets.

The *Refined C-Program* is then supplied to *AFL* along with random *Seeds*. *AFL* produces Test Inputs (crashes, hangs, and queue) and Fuzz Statistics (fuzzer_stats, fuzz_bitmap, and plot_data). Next, *Crash Triage* which is an *AFL* utility to produce the detailed log with all *Crash Details*. AFL might produce unique crashes but, a crash could be reported from different paths, hence the crash details might have duplication of crashes. The component *Unique Target Extractor* searches for the crashes caused by the target located at the unique line in *Refined C-Program*. If such targets detected by *AFL* then they are *Reachable targets*. Also, *Unique Target Extractor* takes the list of alarms that have been already proved as *Unreachable targets*. This component adds both *Reachable targets* and *Unreachable targets* and calls them as *Known targets*. Since this component takes original *C-Program* as input, where the total number of targets exists, so a total number of *UnKnown targets* can be computed.

For example, we consider a program named *sample.c* (22 LOCs) as shown in Listing 1. There are a total of *5 targets* (injected as Div-by-Zero error) at line numbers *8, 10, 13, 17, and 20* as shown in Listing 1. This *sample.c* program is supplied into *Frama-C* and a report is generated as shown in Listing 2. If we observe the report then we can see that *Frama-C* reports an alarm of *division-by-zero* at *line 8* for *sample.c* program. Since, *Frama-C* is a sound static analyser it means that the other injected targets are confirmed unreachable. Now, we will use our proposed and implemented *Code Refiner* to reproduce the program with only alarmed targets as shown in Listing 3 named as *sample-refined.c*. So, there is only *1 alarmed target* at *line number 8*, it means other targets can be disabled or commented for further execution (*4 targets* got commented). Now this *sample-refined.c* program is supplied into *AFL* to produce report. We ran *sample-refined.c* using *AFL* for *600 sec* the target enabled at *line number 8* as *assert(0)* has been detected as a crash. So, finally there were 5 targets out of which *5 targets* have known status, *4*

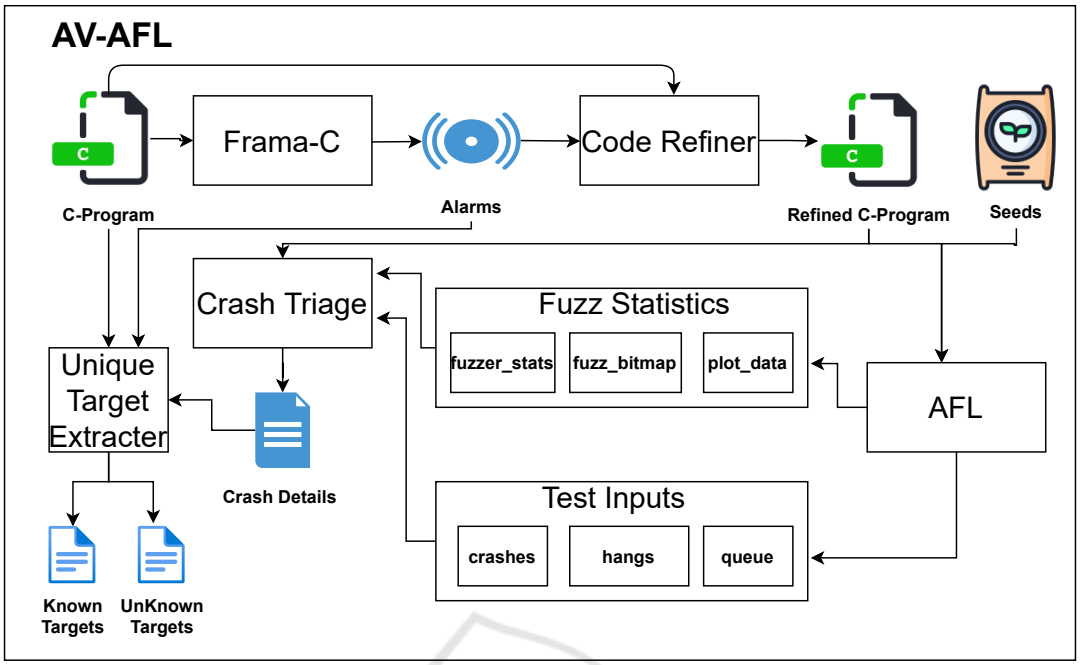


Figure 1: Framework for AV-AFL.

targets are unreachable and *1 target* is reachable as shown in Listing 4. There were no targets which have unknown status, so in this manner fuzzing can be considered as complete.

Listing 1: A sample program: sample.c.

```
#include <stdio.h>
#include <assert.h>
int main(){
 int arg1, arg2;
 int kappa = 0;
 scanf("%d",&arg1);
 scanf("%d",&arg2);
 kappa = kappa / 0; kappa = 0;
 if (arg1 < arg2 && arg2 > 90) {
  kappa = kappa / 0; kappa = 0;
  printf("The value of arg2 is
       greater than 90");
  if (kappa < 0 ) {
   kappa = kappa / 0; kappa = 0;
   if (arg1 < 0) {
    printf("The value of arg1 is
         negative");}
   else {
    kappa = kappa / 0; kappa = 0;
    printf("The value of arg1 is
         positive");}}
  else {
   kappa = kappa / 0; kappa = 0;
   printf("The value of kappa is not
        a negative number");}
 }
 return 0;
}
```

Listing 2: Fram-C report for sample.c.

```
...
[eva:alarm] sample.c:8: Warning:
    division by zero. assert 0 != 0;
...
```

Listing 3: A refined program sample-refined.c.

```
#include <stdio.h>
#include <assert.h>
int main(){
 int arg1, arg2;
 int kappa = 0;
 scanf("%d",&arg1);
 scanf("%d",&arg2);
 assert(0);
 if (arg1 < arg2 && arg2 > 90) {
  //assert(0);
  ...
```

Listing 4: AFL report for sample-refined.c.

```
**Final Result Report from AFL**
Total number Injected Errors =:5
Total number Unreachable Errors =:4
Total number Detected Errors =:1
Total number Undetected Errors =:0
```

4.2 Algorithmic Description

In this section, we explain the algorithmic description of *AV-AFL*. We generate a report for the input program *P*. We supply the program to the fuzzer after eliminating the unreachable vulnerabilities and con-

sidering only alarmed vulnerabilities. In this way the generated program is an improvised program in terms of number vulnerabilities that are to be processed. The Algorithm 1 shows the reporting of Known Targets (*#KT*) and UnKnown Targets (*#UT*) for the input C-Program (*P*) to *AV-AFL*. Line 1 in Algorithm 1 invocation of *Frama-C* which is a static analyser, this gives information of the Alarmed Vulnerabilities (*#AV*). Line 2 in Algorithm 1 shows the conversion of the input C-Program i.e. P into P' using CODE_REFINER component that takes P and *#AV* as inputs. Line 3 in Algorithm 1 invokes American Fuzzy Lop (*AFL*) tool by supplying P' along with Random Seeds (*RS*) generated using random seed generation procedure, it then produces Fuzz Statistics (*FS*) and Test Suite (*TS*). Line 4 in Algorithm 1 calls CRASH_TRIAGE component by supplying P', *FS*, and *TS* and produces Crash Details Log (*CLog*). Line 4 in Algorithm 1 invokes Unique_Target_Extracter component by supplying *P*, *#AV*, and *CLog* to produce *#KT*, and *#UT*. Finally, *#KT* and *#UT* are returned at Line 6 in Algorithm 1. The Algorithm 2

Algorithm 1: AV-AFL.

```
Input: P, RS
Output: {#KT,#UT}
1: #AV ← FRAMAC(P)
2: P' ← CODE_REFINER(P,#AV)
3: {FS,TS} ← AFL(P',RS)
4: CLog ← CRASH_TRIAGE(P',FS,TS)
5: {#KT,#UT} ←
   Unique_Target_Extracter(P,#AV,CLog)
6: return {#KT,#UT}
```

Algorithm 2: CODE_REFINER.

```
Input: P,#AV
Output: {P'}
1: LINES ← LINE-EXTRACTION(#AV)
2: IP ← REMOVAL(P)
3: P' ← INCLUSION(IP,LINES)
3: return {P'}
```

shows the working of intermediate CODE_REFINER component which takes input as C-Program i.e. *P* and Alarmed Vulnerabilities *#AV*. From *#AV* it extracts the line number of the vulnerabilities and outputs a list of line number *LINES* in Line 1. In Line 2 the algorithm removes all the targets from the original C-program using the REMOVAL function and outputs a new intermediate C-program *IP*. In Line 3 it takes the intermediate C-program *IP* and the output of line 1 *LINES* and include out all those lines from *IP* using the INCLUSION function and outputs a final C-program P'.

5 EXPERIMENTAL RESULTS

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

5.1 The Set Up

We used an Intel Core i5-1135G7 CPU @ 2.40GHz Linux box (32-bit Ubuntu 16.04) with 2 GB RAM. All the input programs considered for our study are written in ANSI-C format. For result comparison, we consider *AFL* as our baseline because it is a state-of-the-art tool. We have provided **1 Hr** time budget to each program for running AFL and AV-AFL. the evaluation of the result is based on the guidelines mentioned in "Evaluating Fuzz Testing" (et al.,). It is to be noted that Frame-C and Code Refiners were completed in approx 1 minute only on an average of 45 programs which is negligible. We considered Random seed for AFL and AV-AFL.

5.2 Benchmarks Tested

Reactive systems appear everywhere, e.g. as Web services, decision support systems, or logical controllers. Since the approach mentioned requires a C-program therefore we are considering RERS programs that replicate the real-world applications from Avionics, Banking, Medical, Railways, etc. They are from RERS challenge competition in years 2019 (RERS, 2019b; RERS, 2019a), and 2020 (RERS, 2020). These programs are from the small and moderate size group and easy to hard categories. The codes contain a lot of Boolean expressions, plain assignments, arithmetic operations, and data structures.

5.3 Working Example

We take *Problem18-R20* program from our experimental study reported in Sections 5.4 and 5.5.

Problem18-R20 is supplied to *AFL* and *AV-AFL* both to observe the status of targets. The *AFL* has Unreachable Targets as **0** because there is no such mechanism in fuzzing to prove non-reachabilty of a target. Thus it has to explore for all the targets in original program given. In case of *AFL* the search space is large in comparison to *AV-AFL*. It is observed that out of **100** targets, *AFL* is able to detect only **18** unique crashes in **1 Hr** time budget. Thus, making the total **Known Targets** as **18** and **Unknown Targets** as **82**. On the other hand, it is observed that total targets

Table 1: Experimental results on 45 RERS programs.

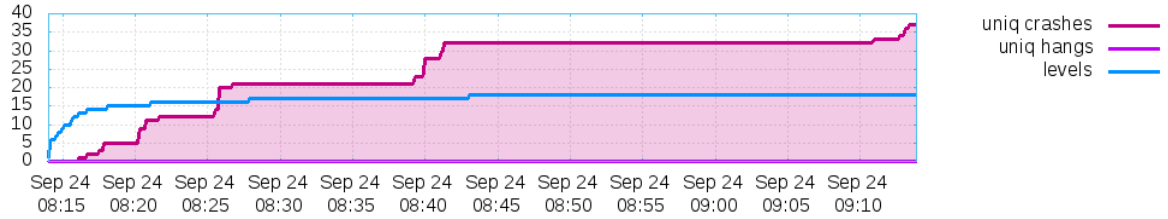
Programs	LOCs	#Targets	AFL				AV-AFL			
			#Unreachable Targets	#Reachable Targets	#Known Targets	#UnKnown Targets	#Unreachable Targets	#Reachable Targets	#Known Targets	#UnKnown Targets
m22_Reach	5002	100	0	14	14	86	0	12	12	88
m24_Reach	23125	100	0	4	4	96	0	6	6	94
m27_Reach	18645	100	0	2	2	98	6	5	11	89
m41_Reach	3144	100	0	65	65	35	3	43	46	54
m45_Reach	14344	100	0	15	15	85	0	8	8	92
m49_Reach	18680	100	0	17	17	83	0	18	18	82
m54_Reach	2554	100	0	79	79	21	0	88	88	12
m55_Reach	19721	100	0	0	0	100	3	1	4	96
m_76Reach	18620	100	0	14	14	86	3	14	17	83
m_95Reach	3500	100	0	9	9	91	8	8	16	84
m106_Reach	4197	100	0	1	1	99	0	1	1	99
m131_Reach	88800	100	0	7	7	93	0	7	7	93
m135_Reach	2989	100	0	2	2	98	4	2	6	94
m158_Reach	2048	100	0	9	9	91	5	12	17	83
m159_Reach	2328	100	0	9	9	91	0	9	9	91
m164_Reach	2482	100	0	31	31	69	6	24	30	70
m167_Reach	7719	100	0	1	1	99	10	8	18	82
m172_Reach	6083	100	0	4	4	96	3	31	34	66
m173_Reach	55859	100	0	20	20	80	1	22	23	77
m181_Reach	522136	100	0	MO	0	100	0	MO	0	100
m182_Reach	142430	100	0	10	10	90	1	9	10	90
m183_Reach	1656	100	0	70	70	30	1	71	72	28
m185_Reach	13215	100	0	0	0	100	3	0	3	97
m189_Reach	42707	100	0	0	0	100	1	0	1	99
m190_Reach	192855	100	0	12	12	88	0	11	11	89
m196_Reach	10444	100	0	74	74	26	1	84	85	15
m199_Reach	2358	100	0	7	7	93	1	27	28	72
problem11-R19	1143	100	0	15	15	85	56	16	72	28
problem12-R19	2061	100	0	0	0	100	45	0	45	55
problem13-R19	1877	100	0	14	14	86	49	14	63	37
problem14-R19	4691	100	0	24	24	76	53	24	77	23
problem15-R19	13213	100	0	0	0	100	15	0	15	85
problem16-R19	88617	100	0	0	0	100	0	0	0	100
problem17-R19	17342	100	0	39	39	61	34	38	72	28
problem18-R19	61608	100	0	0	0	100	11	0	11	89
problem19-R19	793391	100	0	MO	0	100	0	MO	0	100
problem-11-R20	1168	100	0	17	17	83	68	17	85	15
problem-12-R20	2298	100	0	0	0	100	49	0	49	51
problem-13-R20	2190	100	0	19	19	81	27	19	46	54
problem-14-R20	4183	100	0	4	4	96	46	4	50	50
problem-15-R20	26205	100	0	39	39	61	16	38	54	46
problem-16-R20	113733	100	0	11	11	89	2	11	13	87
problem-17-R20	18040	100	0	30	30	70	38	30	68	32
problem-18-R20	127848	100	0	18	18	82	19	27	46	54
problem-19-R20	518567	100	0	MO	0	100	0	MO	0	100

for *Problem18-R20* are **100**, out of which the targets are present only at **81** locations, thus bring down the unreachable targets to **19**. Thus the search space for fuzzer reduces from **100** to **81** Out of these **81** targets *AV-AFL* is able to detect crashes for **27** targets in **1 Hr** of given time budget. Thus making the total **Known Targets** as **46** and **Unknown Targets** as **54**.

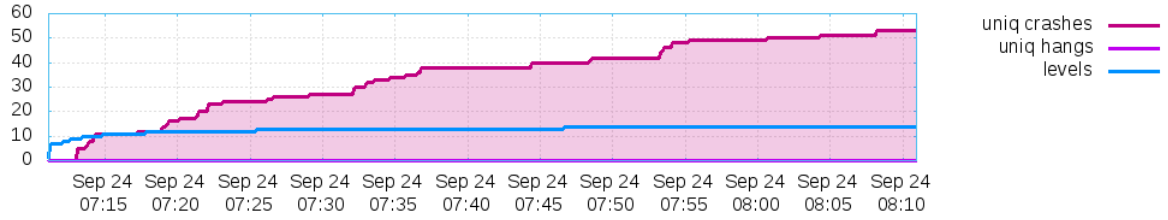
Fig. 2 shows the crash report charts for *Problem18-R20* program using *AFL* and *AV-AFL*. As we already discussed the proving of unreachable targets using a sound static analyser, we claim that the task overhead gets reduced once we refine the program by removing the unreachable targets and only searching for alarmed targets. Figures 2a and 2b show the crash status over the time of 1 Hr. Since, the searching for targets are less in *AV-AFL* so the speed of fuzzing is higher as compared to *AFL* as shown in Figures 2a and 2b. For example, if we consider midtime of the fuzzing i.e. *30 minutes* so we can observe that *AFL* identifies **32 uniq crashes**. It is to be noted that these uniq crashes have the redundancy from different paths, the final unique reachable targets have been reported in Sections 5.4 and 5.5, whereas *AV-AFL* identifies **40 uniq crashes** These results show our claims for *AV-AFL*.

5.4 Results Evaluation

Table 1 shows the experimental results on 45 RERS programs we tested. The Column1 shows the name of programs. The programs with suffix **_Reach* are taken from *Industrial Reachability Problems, RERS-2019*. The programs with suffix **-R19* are taken from Sequential Reachability Problems, RERS-2019. The



(a) AFL



(b) AV-AFL

Figure 2: Crash report charts for Problem18-R20.

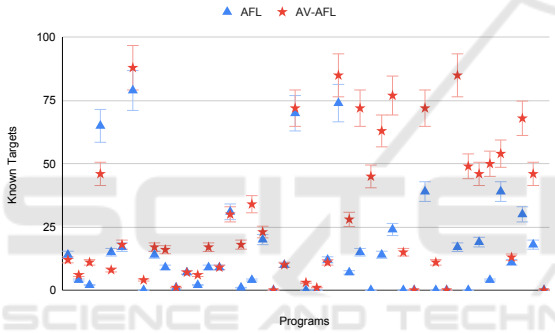


Figure 3: Results showing the Known Targets.

programs with suffix **-R20* are taken from Sequential Reachability Problems, RERS-2020. Column2 shows the size of programs in LOCs. This ranges from **1143** to **793391**. Column3 shows number of targets present in the programs. We can observe that all the programs have **100** targets in each program. The Columns 4 and 5 are the results for AFL and AV-AFL. Both columns further divided into four columns viz. *#Unreachable*, *#Reachable*, *#Known*, and *#UnKnown* targets.

Column *#Unreachable* Targets under AFL has **0** targets, because traditional AFL has no capabilities to prove the unreachable targets, because fuzzing in principle cannot finish the execution. Columns *#Reachable* and *#Known* Targets under AFL has total number of unique crashes (it means the actual line number identified where the targets were existing). Column *#UnKnown* targets under AFL can be computed from #Targets - *#Known* targets.

Similarly, Column *#Unreachable* Targets under AV-AFL have been computed from the Sound Static Analyser i.e. Frama-C (#Targets - #Alarmed Targets). There are **32 out of 45** programs for which Frama-C has proved > 0 unreachable targets. Column *#Reachable* Targets under AV-AFL have been identified as actual unique crashes after eliminating *#Unreachable* Targets from the programs. Column *#Known* Targets under AV-AFL have been computed as #Unreachable + #Reachable Targets. *#UnKnown* targets under AV-AFL can be computed from #Targets - *#Known* targets under AV-AFL.

5.5 Discussion on Results

There are total **32 out of 45** programs (Highlighted with green color in Table 1 to show winning situations) for which our proposed approach AV-AFL has more number of #Known Targets as compared to the baseline AFL. Fig. 3 shows the results of known targets for AFL and AV-AFL. It can be observed that red-stars (AV-AFL) dominate blue-triangles (AFL) in terms of showing the known status of the targets present in programs. There are **29** out of these 32 programs where AV-AFL had at least 1 *#Unreachable* Target. Also, there are **6** out of 32 programs where AV-AFL had at least 1 *#Unreachable* Target. AV-AFL has more #Reachable Targets for 14 programs, however, for 19 programs the #Reachable Targets were the same as compared to AFL. This shows the clear picture that AV-AFL is superior. There are **3 out of 45** programs for which both AFL and AV-AFL have Memory Out (MO) error, so we could not prove any targets hence all were *#UnKnown* targets. For the rest programs, AV-AFL has ineffective results.

There were **45** programs with **4500** existing targets. For AFL, in total **706** targets were known and **3794** targets were unknown. Next, AV-AFL shows

total **1347** targets as known and **3153** unknown. Finally, AV-AFL is able to show the known status for **641** extra targets in comparison to AFL.

The main drawback of AFL remains its inability to distinguish and segregate the unreachable vulnerabilities from the group of vulnerabilities. We have experimentally proven that the AV-AFL overcomes this inability successfully by alarming only the reachable vulnerabilities. It has been observed through results that the performance of AV-AFL for the same time period is highly improved in comparison to AFL due to reduced search space.

6 CONCLUSION

The AV-AFL approach presented in this paper facilities the smart detection of crashes by eliminating the unreachable targets by the fuzzing mechanism. It has been observed from the literature of the fuzzing domain that vulnerability detection is a forever running process. Use of fuzzing enables us to detect the vulnerabilities so that attackers may not misuse them to exploit the system. But, this detection will continue because of the unknown status of the vulnerabilities. If fuzzer could be able to show that the vulnerabilities it is searching for are not required and the time can be spent on other vulnerabilities which could lead to a crash, then the fuzzer will be efficient to perform the fuzzing fast. AV-AFL provides this environment using sound static analyzer Frama-C. It is experimentally observed that the proposed AV-AFL detects the vulnerabilities effectively in comparison to the baseline AFL. In total AV-AFL shows **641** extra as known targets in contrast to AFL. AV-AFL has better results in total **71.11% of 45** programs. It shows that AV-AFL is superior.

In the future, we will extend AV-AFL with a new seed generation technique to improvise the vulnerability detection process. We will try to embed the model checker technique with AFL to prove the unknown cases at last.

REFERENCES

Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., et al. (2021). The dogged pursuit of bug-free c programs: the frama-c software analysis platform. *Communications of the ACM*, 64(8):56–68.

Böhme, M., Pham, T., Nguyen, M.-D., and Roychoudhury, A. (2017). Directed greybox fuzzing. pages 2329–2344.

et al., K. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

Godboley, S., Dutta, A., Mohapatra, D. P., Das, A., and Mall, R. (2016). Making a concolic tester achieve increased mc/dc. *Innovations in systems and software engineering*, 12(4):319–332.

Godboley, S., Dutta, A., Mohapatra, D. P., and Mall, R. (2017a). J3 model: a novel framework for improved modified condition/decision coverage analysis. *Computer Standards & Interfaces*, 50:1–17.

Godboley, S., Dutta, A., Mohapatra, D. P., and Mall, R. (2018a). Gecojap: A novel source-code preprocessing technique to improve code coverage. *Computer Standards & Interfaces*, 55:27–46.

Godboley, S., Dutta, A., Mohapatra, D. P., and Mall, R. (2018b). Scaling modified condition/decision coverage using distributed concolic testing for java programs. *Computer Standards & Interfaces*, 59:61–86.

Godboley, S., Jaffar, J., Maghareh, R., and Dutta, A. (2021). Toward optimal mc/dc test case generation. ISSTA 2021, page 505–516, New York, NY, USA. Association for Computing Machinery.

Godboley, S., Mohapatra, D. P., Das, A., and Mall, R. (2017b). An improved distributed concolic testing approach. *Software: Practice and Experience*, 47(2):311–342.

Godboley, S., Sahani, A., and Mohapatra, D. P. (2015). Abce: A novel framework for improved branch coverage analysis. In *SCSE*, pages 266–273.

Iorga, D., Corlătescu, D., Grigorescu, O., Săndescu, C., Dascălu, M., and Rughiniş, R. (2020). Early detection of vulnerabilities from news websites using machine learning models. In *RoEduNet*, pages 1–6. IEEE.

Kiss, B., Kosmatov, N., Pariente, D., and Puccetti, A. (2015). Combining static and dynamic analyses for vulnerability detection: Illustration on heartbleed. In Piterman, N., editor, *Hardware and Software: Verification and Testing*, pages 39–50, Cham. Springer.

RERS (2019a). RERS19:Industrial Reachability Problems. http://rers-challenge.org/2019/index.php?page=industrialProblemsReachability.

RERS (2019b). RERS19:Sequential Reachability Problems. http://rers-challenge.org/2019/index.php?page=reachProblems.

RERS (2020). RERS20:Sequential Reachability Problems. http://rers-challenge.org/2020/index.php?page=reachProblems.

Rustamov, F. e. a. (2021). Bugminer: Mining the hard-to-reach software vulnerabilities through the target-oriented hybrid fuzzer. *Electronics*, 10(1).

Shastry, B. e. a. (2017). Static program analysis as a fuzzing aid. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 26–47. Springer.

Signoles, J. (2021). The e-acsl perspective on runtime assertion checking. VORTEX 2021, page 8–12. Association for Computing Machinery.