Towards Complete Fuzzing with KLEE

Kanika Gupta, Sangharatna Godboley

NITMINER Technologies,
Department of Computer Science and Engineering,
National Institute of Technology, Warangal, Warangal, Telangana, India,
kanikagupta.gupta18@gmail.com, sanghu@nitw.ac.in

KLEE Workshop 2024

Co-Organised with
46th International Conference on Software Engineering (ICSE 2024) ,
15-16 April, 2024,
Lisbon, Portugal

April 10, 2024

Plan of the talk

1. Objective
2. Introduction
3. Proposed Idea
4. Implementation Details
5. Experimental Study
6. Result Analysis
7. Conclusion
8. References

Objective

Comp-AFL = Framma-C + AFL + KLEE

Introduction

- The software can only be considered safe for release and use once all bugs and vulnerabilities have been identified and eliminated.
- Fuzzing is a technique within the software testing domain that can be employed to detect vulnerabilities.
- However, fuzzing does have certain drawbacks, such as speed, coverage, and efficiency issues.
- Traditional fuzzers often struggle to efficiently identify software vulnerabilities within a given timeframe.
- In this paper, we present a comprehensive fuzzing technique designed to enhance the effectiveness of fuzzers, particularly in terms of identifying the status of targets within the software. This technique, referred to as Complete AFL (Comp-AFL), aims to detect more vulnerabilities, thus advancing the concept of complete fuzzing.

Introduction

- Comp-AFL introduces a method by which the fuzzer can identify a greater number of vulnerabilities, bringing it closer to achieving the goal of complete fuzzing.
- This approach streamlines the process by leveraging the static analysis tool Frama-C to eliminate the extra time that fuzzers typically spend exploring unreachable vulnerabilities.
- Furthermore, we enhance the efficiency using the dynamic symbolic tool KLEE, which identifies additional known targets to optimize the fuzzing process.
- Our experimental results demonstrate that the proposed Comp-AFL approach consistently outperforms both baseline AFL and AV-AFL across all 40 programs, achieving superior results in **100%** of cases.
- Notably, Comp-AFL achieves a state of complete fuzzing by identifying all targets as known targets in **25% of the 40** programs.

Proposed Idea

- The framework of **Comp-AFL**, as depicted in Figure 1, integrates static analysis of the source code with a fuzzing strategy followed by dynamic symbolic execution to efficiently detect vulnerabilities.
- The process unfolds as follows:
  1. The original C-Program is supplied to Frama-C, a sound static analyzer, which utilizes the EVA plug-in to extract alarm details from the program. Alarmed vulnerabilities' locations become the targets for fuzzing.
  2. Non-alarmed targets in the C-Program are proven as Unreachable targets.
  3. These unreachable targets are combined with the number of Known targets.

Proposed Idea

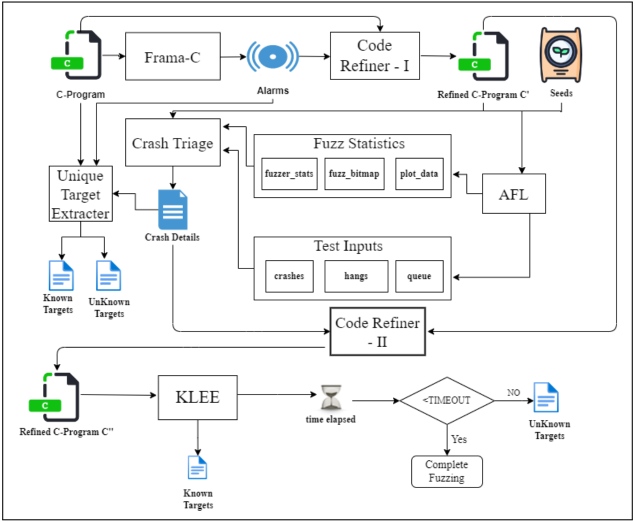

Figure 1: Framework for Comp-AFL

Implementation Details

- The Code Refiner-I component takes the C-Program and the list of Alarms to produce a Refined C-Program.
    - This refined version focuses solely on the meaningful targets of interest for fuzzing.
    - The Refined C-Program is then fed into AFL along with random Seeds.
    - AFL generates Fuzz Statistics (fuzzer_stats, fuzz_bitmap, plot_data) and Test Inputs (crashes, hangs, queue).

Implementation Details

- Crash Triage, an AFL utility, produces a detailed log with Crash Details (CLog).
    - The Unique Target Extractor component identifies unique crashes within the Refined C-Program using the information from AFL.
    - These targets are marked as Reachable targets.
    - Additionally, the Unique Target Extractor considers the list of alarms already proven as Unreachable targets by Frama-C.
    - Both Reachable and Unreachable targets are combined and termed as Known targets (#K).
    - The number of Unknown targets (#UK) can be calculated by subtracting the number of Known targets from the total number of targets in the original program.

Implementation Details

- To further reduce the number of unknown targets, dynamic symbolic execution is introduced.
  - As the crash details obtained from AFL already include identified crashes and their line numbers, the proposed method removes these crashes from the process and adds them to #K.
  - Code Refiner-II utilizes the crash information to eliminate these crashes from the refined code, producing refined-code-II.
  - Refined-code-II is supplied to KLEE, a dynamic symbolic execution tool.

Implementation Details

- KLEE processes the refined C program and detects crashes within a certain execution time, resulting in #K targets and an Elapsed Time.
    - If KLEE execution completes within the TIMEOUT, all targets are considered Known (since KLEE is a sound tool), achieving complete fuzzing with zero Unknown targets.
    - However, if KLEE execution exceeds TIMEOUT, it implies the presence of Unknown targets that require further investigation to achieve complete fuzzing.
    - The TIMEOUT can be adjusted based on the time budget.
    - Absolute complete fuzzing can be achieved by running the program with an infinite time budget.

Experimental Study

- We conducted our experiments on a Linux system running a 64-bit Ubuntu 16.04 distribution, equipped with an Intel Core i5-1135G7 CPU operating at 2.40GHz and 4.8 GB of RAM.
- Our benchmark for the experimentation consisted of 40 RERS programs, chosen to encompass a broad range of difficulty and complexity levels.
- These programs offer a representative spectrum of real-world applications, including domains such as Avionics, Banking, Medical, and Railways, among others [2,3,4,5].
- Detailed experimental data and accompanying code scripts can be found in the provided artefacts [1].

Experimental Study

Table 1: Experimental results on 40 RERS programs

Programs	LOCs	AFL					AV-AFL				Comp-AFL							
		#Tr	#U	#R	#K	#UK	#U	#R	#K	#UK	afl-#U	afl-#R	klee	klee-#D	TE	#K	#UK
m22_Reach	5002	100	0	14	14	86	0	12	12	88	0	9	91	17	0:30:17	26	74
m24_Reach	23125	100	0	4	4	96	0	6	6	94	0	4	96	6	0:30:23	10	90
m27_Reach	18645	100	0	2	2	98	6	5	11	89	6	4	90	6	0:30:17	16	84
m41_Reach	3144	100	0	65	65	35	3	43	46	54	3	66	31	9	0:30:26	78	22
m45_Reach	14344	100	0	15	15	85	0	8	8	92	0	11	89	10	0:30:42	21	79
m49_Reach	18680	100	0	17	17	83	0	18	18	82	0	18	82	3	0:30:19	21	79
m54_Reach	2554	100	0	79	79	21	0	88	88	12	0	82	18	7	0:03.01	100	0
m55_Reach	19721	100	0	0	0	100	3	1	4	96	3	1	96	0	0:30:16	4	96
m_76Reach	18620	100	0	14	14	86	3	14	17	83	3	14	83	0	30.13	17	83
m_95Reach	3500	100	0	9	9	91	8	8	16	84	8	8	84	17	30.41	33	67
m106_Reach	4197	100	0	1	1	99	0	1	1	99	0	2	98	2	30.12	4	96
m135_Reach	2989	100	0	2	2	98	4	2	6	94	4	3	93	5	0:25.66	100	0
m158_Reach	2048	100	0	9	9	91	5	12	17	83	5	7	88	22	0:30:13	34	66
m159_Reach	2328	100	0	9	9	91	0	9	9	91	0	14	86	15	0:30:12	29	71
m164_Reach	2482	100	0	31	31	69	6	24	30	70	6	31	63	30	0:30:12	67	33
m167_Reach	7719	100	0	1	1	99	10	1	11	89	10	1	89	1	0:30:18	12	88
m172_Reach	6083	100	0	3	3	97	3	6	9	91	3	3	94	4	0:30:12	10	90
m173_Reach	55859	100	0	3	3	97	1	3	4	96	1	3	96	0	0:30:10	4	96
m182_Reach	142430	100	0	3	3	97	1	4	5	95	1	3	96	1	0:30:10	5	95
m183_Reach	1656	100	0	65	65	35	1	64	65	35	1	62	32	5	0:05:09	100	0
m185_Reach	13215	100	0	0	0	100	3	0	3	97	3	0	97	0	0.30:17	3	97
m189_Reach	42707	100	0	0	0	100	1	0	1	99	1	0	99	3	0:30:12	4	96
m190_Reach	192855	100	0	12	12	88	0	11	11	89	0	11	89	3	30.09	14	86
m196_Reach	10444	100	0	14	14	86	1	16	17	83	1	16	83	0	0:30:05	17	83
m199_Reach	2358	100	0	28	28	72	1	27	28	72	1	26	73	7	0:30:20	34	66
problem11-R19	1143	100	0	15	15	85	56	16	72	28	56	16	28	0	0:01:51	100	0
problem12-R19	2061	100	0	0	0	100	45	0	45	55	45	0	55	0	0:02:43	100	0
problem13-R19	1877	100	0	14	14	86	49	14	63	37	49	14	37	0	0:03:06	100	0
problem14-R19	4691	100	0	24	24	76	53	24	77	23	53	24	23	0	0:30:02	77	23
problem15-R19	13213	100	0	0	0	100	15	0	15	85	15	0	85	0	0:30:17	15	85
problem17-R19	17342	100	0	39	39	61	34	38	72	28	34	36	30	3	0:30:21	73	27
problem18-R19	61608	100	0	0	0	100	11	0	11	89	11	0	89	0	0:30:15	11	89
problem-11-R20	1168	100	0	17	17	83	68	17	85	15	68	17	15	0	0:01:43	100	0
problem-12-R20	2298	100	0	0	0	100	49	0	49	51	49	0	51	0	0:03:32	100	0
problem-13-R20	2190	100	0	19	19	81	27	19	46	54	27	19	54	0	0:03:05	100	0
problem-14-R20	4183	100	0	4	4	96	46	4	50	50	46	4	50	0	0:28:08	100	0
problem-15-R20	26205	100	0	0	0	100	16	0	16	84	16	0	84	0	0:30:19	16	84
problem-16-R20	113733	100	0	1	1	99	2	1	3	97	2	1	97	0	0:30:18	3	97
problem-17-R20	18040	100	0	30	30	70	38	30	68	32	38	30	32	0	0:30:28	68	32
problem-18-R20	127848	100	0	0	0	100	19	0	19	81	19	0	81	0	0:30:13	19	81

KLEE Workshop 2024

Result Analysis

- The columns *#U* represents the number of Unreachable targets; *#R*, indicating the number of reachable targets; *#K*, denoting the total count of known targets; and *#UK*, which signifies the total number of unknown targets.
- *afl-#U*, which represents the count of unreachable targets detected by AFL; *afl-#R*, indicating the number of reachable targets detected by AFL; *klee*, which represents the targets provided as input to Klee; *Klee-#D*, representing the count of targets detected by Klee; *TE*, indicating the elapsed time of Klee's execution; *#K*, representing the known targets; and *#UK*, denoting the unknown targets.

Result Analysis

- Column $\#U$ under AFL contains **0** targets, as traditional fuzzers are unable to detect unreachability due to their inability to complete execution.
- The $\#R$ column shows the number of targets detected by AFL as unique crashes, and the $\#K$ column represents the total count of targets whose status is now known, computed as the sum of $\#U$ and $\#R$.
- Similarly, the $\#U$ column under AV-AFL represents the number of unreachable targets, calculated using Frama-C, a sound static analysis tool.
- The value of $\#UR$ is derived from **(#Tr - #Alarmed Targets)**. Notably, in **32 out of 40** programs, Frama-C has proven the presence of more than zero unreachable targets.

Result Analysis

- The column *afl-#U* within Comp-AFL is derived from the Sound Static Analyser.
- The column *afl-#R* for targets under Comp-AFL represents the actual unique crashes identified after removing *#U* targets from the programs during AV-AFL execution.
- The column *Klee* denotes the search space for Klee's execution, which is calculated using the formula **#Tr - afl-#U + afl-#R** for targets.
- The column *#K* represents targets under Comp-AFL and is computed as **afl-#U + afl-#R + Klee-#D**. The *#UK* column, indicating unknown targets within Comp-AFL, can be calculated as **#Targets - #K** targets within Comp-AFL.

Conclusion

- Comp-AFL, consistently exhibits a higher number of #K Targets across all tested programs compared to the baseline AFL and AV-AFL.
- Notably, among the 40 tested programs, AV-AFL outperforms AFL in terms of #K Targets in a total of **32 cases** (highlighted in green in Table 1).

Acknowledgement

This work is sponsored by IBITF, Indian Institute of Technology (IIT) Bhilai, under the grant of PRAYAS scheme, DST, Government of India.

References

1. 2022. Experimental Artifacts: `https://figshare.com/s/7053da855851c6c6fd81`.
2. RERS 2012. RERS:. `http://rers-challenge.org/`
3. RERS 2019. RERS19:Industrial Reachability Problems. `http://rers-challenge.org/2019/index.php?page=industrialProblemsReachability`
4. RERS 2019. RERS19:Sequential Reachability Problems. `http://rers-challenge.org/2019/index.php?page=reachProblems`
5. RERS 2020. RERS20:Sequential Reachability Problems. `http://rers-challenge.org/2020/index.php?page=reachProblems`

Thank You!