Trace Simplification to Aid Fuzzing

Nikolas R. L. Tyllis

Supervised by Assoc. Prof. Damith C. Ranasinghe and Mathew Duong

Abstract— Methods of software testing for large scale applications involving manually engineered test cases are prone to human error. Now software testing largely relies on fuzzers. Fuzzers repeatedly mutate a corpus of inputs and save inputs that cause crashes or hangs. These fuzzers generate large amounts of output. Crash deduplication and crash clustering are methods that can be used to process fuzzer output and produce useful data for the bug fixer. Fuzzers can also be more efficient if they are initialised on a minimized but effective corpus. In this paper current state-of-the-art solutions are discussed, a crash clustering tool is evaluated and a corpus minimization tool is evaluated.

Keywords— Fuzz Testing, Software Security, Vulnerability Discovery.

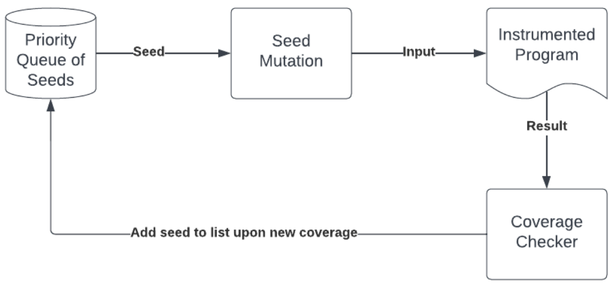
I. Introduction

Fuzzing or fuzz testing is a form of software testing that uses input mutation and generation to discover vulnerabilities and bugs in software that result in crashes or hangs [1]. The process involves an initial seed input that is mutated upon the occurrence of unanticipated crashes or in some coverage guided fuzzers, upon an increase of code coverage. The crashing inputs are then recorded such that debugging tools can be used to triage the vulnerabilities.

The concept of fuzzing was first proposed by Miller *et al.* in 1988 when the program initially named *fuzz* was developed [2]. This program led to the discovery that a minimum of a quarter of the utilities on all UNIX systems could be crashed due to inputs generated by *fuzz*. *Fuzz* is an example of the first black box fuzzer. A black box fuzzer generates inputs with no knowledge of the implementation of the software. Although black box fuzzers are effective at finding vulnerabilities they often lead to low code coverage. An example of this is if the conditional statement *if (x == 7)* is restricting access to a significant block this block only has a one in 2^{32} chance of being exercised for a randomly chosen 32-bit value for *x* [3]. This and similar issues led to the development of fuzzers such as LibFuzz and American Fuzzy Lop (AFL) which popularised coverage-guided fuzzing, the latter of which has dominated much fuzzing research since its release in 2013. Coverage-guided fuzzers rely on building a corpus of inputs that have resulted in new code coverage [4]. These inputs are then mutated to find new paths through the target, see Fig. 1.

The fuzzer American Fuzzy Lop Plus Plus (AFL++) is another fuzzer. AFL++ is an open source, coverage-guided fuzzer building upon AFL which has had large use in academia and industry in the past [5]. AFL++ encounters similar limitations to other fuzzers such as low efficiency, low coverage and crash duplication however, despite this it has remained a state-of-the-art vulnerability discovery tool [6].

Fig. 1: High level logic of a coverage-guided fuzzer.



Crash duplication describes the scenario when multiple crashes discovered by the fuzzer have the same root cause bug. Analysing large amounts of crashes generated and manually deduplicating them with debugging tools requires considerable effort and time. AFL++ does have some tools built in such as *afl-cmin* that aims to find the smallest subset of inputs that still trigger the full range of crashes.

Minimization tools such as *afl-cmin* use techniques such as stack hashing for crash deduplication or more commonly for corpus minimization [7]. Stack hashing relies on hashing the information of the call stack at the moment of a crash [8]. These hashes can then be compared and used for deduplication. Another method of crash deduplication is analysing the crash site, such as the value stored in the instruction register at the moment of the crash.

II. Motivation

Software vulnerabilities have become a major vector for cyberspace threats. A vulnerability is a flaw in the design or implementation of a system that could be exploited in a malicious attack [9]. Fuzzing is currently the most popular software vulnerability discovery technique. When compared with other techniques, fuzzing is easy to deploy as it can be performed with or without knowledge of underlying implementation.

Any bug that results in behaviour such as a crash or hang can be utilised in a Denial-of-service (DoS) attack. The DoS attack is a popular method of attack which often results in a large monetary impact. A DoS attack is defined as when a regular user or administrator cannot access or use a service due to a malicious actor [10]. Most crashes or hangs discovered through fuzzing could theoretically be exploited for a DoS attack if they are on the surface of a system with no input sanitation. It is important to develop and evaluate methods that developers can use when fuzz testing their software to increase efficiency and so that they have to manually analyse the least amount of data possible.

III. Review

a. Igor: Crash Deduplication Through Root-Cause Clustering [11]

Igor is a crash deduplication tool that uses program flow to cluster bugs by their root cause. Igor works by minimizing the path length of each bugs execution trace this results in pruned test cases that exercise the behavior necessary for triggering a bug. Then a graph similarity comparison is used to cluster crashes based on the control-flow graph of the minimized execution traces, with each cluster mapping back to a single, unique root cause.

When Igor was evaluated against 39 bugs resulting from 254,000 crashing inputs, distributed over 10 programs. Igor accurately grouped these crashes into 48 uniquely identifiable clusters. Other state-of-the-art tools yielded clusters of almost an entire magnitude higher.

This paper highlights the shortcomings of regular crash bucketing methods such as stack hashing, crash site and coverage profiling. It proposes a counterpart approach of coverage minimization to prune the program flow and minimize the path to a root cause so that comparisons can be performed easier.

b. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation [12]

AURORA is an approach to automatically pinpoint the location of the root cause and provide an explanation of the crash. The goal of AURORA was to identify and explain the root cause of highly exploitable bug classes such as type confusions, use-after-free vulnerabilities and heap buffer overflows. It functions by utilising AFL's built in crash exploration mode that keeps fuzzing and saving crashing inputs until they no longer crash the program to generate a set of inputs for a single crash. A pintool was then implemented that relies on Intel's architecture-specific inspection APIs to monitor the input behaviour. An explanation synthesizer was also developed to isolate behavior in the form of predicates that correlate to differences between crashing and non-crashing runs. These predicates are used to identify problematic code locations somewhere on the path between root cause and crashing location. These are made to aid an analyst in triaging the bug.

The evaluation showed that AURORA was able to able to uncover root causes even for complex bugs and in contrast to existing approaches, AURORA was also able to handle bugs such as type confusion bugs and null dereferences that usually require a data dependency between root cause and crash.

This paper has a unique approach to crash analysis that can be applied to bugs such as type confusions and null dereferences that have been historically hard to manually debug [13] equipping developers with a valuable tool for automated root cause explanation.

c. MoonLight: Effective Fuzzing with Near-Optimal Corpus Distillation [14]

Moonlight is a corpus distillation and minimization tool used to decrease the number of files in a fuzzer input corpus as well as decrease the size of these files while maintaining code coverage and unique program behaviour.

Moonlight was structured around four key principles. The coverage of each different target behaviour should be maximised such that unique paths are covered to their full extent. Behaviour that is repeated across different seeds is eliminated such that it is represented by a single seed. The total size of the input corpus is minimized. The sizes of the seed during fuzzing should also be kept at a minimum to reduce I/O requests on the storage system.

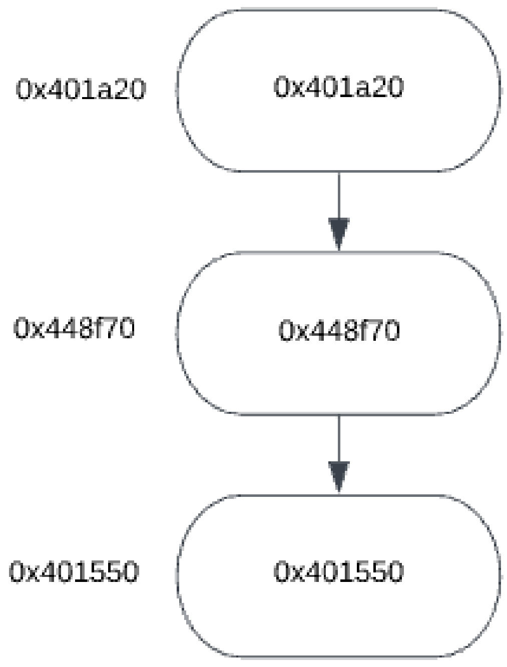
Moonlight provides three different algorithms for corpus distillation. One was the main algorithm developed using coverage matrix analysis, another was this algorithm weighted by execuation time and another was this algorithm weighted by file size.

The Moonlight corpus distillation tool proved to be ineffective during evaluation against the current state of the art corpus minimisation tool afl-cmin.

IV. Trace Generation

The traces are generated via a binary instrumentation tool that uses Intel's Pin API to instrument each of the basic blocks with an address. These traces are produced via a program run using each of the crashing inputs from the fuzzer output or sample, or they can be used to generate traces for the fuzzer input corpus to be used in minimisation. These traces map out the control flow of a program run for further analysis, see Fig. 2.

Fig. 2: Trace output and control flow graph example.



V. Trace Simplification

In order for the edge count data to meaningfully represent the control flow of each crash run the traces need to be simplified. This process involves two steps. First any blocks that are repeated one after each other, such as those of a loop are collapsed into one occurrence. This is achieved by running the bash command uniq, see Fig. 3.

Fig. 3: A trace before and after running the uniq command.

```
0x401a20
0x401a20  0x401a20
0x448f70  0x448f70
0x401550  0x401550
0x401550  0x401a20
0x401550
0x401a20
```

The second step in the trace simplification process involves collapsing multi-line patterns into a single iteration. This occurs if there are branches in each loop iteration resulting in repeated multi-block trace data. This process is run repeatedly on each file until they no longer decrease in size. This is due to block data from embedded loops, see Fig. 4.

Fig. 4: A trace as it undergoes multi-line simplification.

```
Original  Pass 1    Pass 2
0x401a20
0x448f70  0x401a20
0x401550  0x448f70
0x448f70  0x401550  0x401a20
0x401550  0x401a20  0x448f70
0x401a20  0x448f70  0x401550
0x448f70  0x401550
0x401550
```

VI. Trace Processing

The first step of trace processing is to remove duplicate traces. This is simply done through size and hash comparisons. This aims to remove the issues of multiple bugs being in the same cluster leading to a higher cluster purity furthermore, this decreases the workload for the clustering process. This can also be used for corpus minimisation if the traces are from the initial fuzzer input corpus.

For root cause clustering an outlier removal script is applied to the trace corpus. This script removes outlying traces using their file sizes. This is to ensure that outliers do not interfere with the clustering process by effecting the number of clusters generated. There is a drawback to this as traces that do not appear commonly may be removed if they are uniquely large or small.

Upon the completion of these steps the traces are ready to be input in the clustering process.

VII. Trace Clustering

The clustering process involves two steps. First the hit count matrices are constructed from the traces. This involves increasing the corresponding adjacency value in the matrix for each occurrence of an edge in the trace file, see Fig. 5.

Once the adjacency matrices are constructed they are clustered using k-means clustering. The k-means clustering algorithm is a vector clustering algorithm that can cluster points in n-dimensional space [15]. To use k-means clustering an optimal k number of clusters needs to be found. This can be done using the silhouette score [16]. The trace clustering tool allows the user to input the number of probable clusters to probe, this value must be less than the number of traces left after trace processing for the clustering to work. The corresponding k value with the largest silhouette score is used for the final number for clusters.

VIII. Research Questions

The goal of the project is to prototype a tools software testers can use to analyse fuzzer output in a more efficient manner while maintaining accuracy and to develop a tool to minimize a fuzzer corpus to increase time-to-coverage. More specifically the following questions are answered:

a. Is the trace processing step effective at minimising workload while maintaining accuracy?

b. Can crashing inputs be effectively clustered by root cause using this method?

c. Can trace simplification and deduplication be used for effective corpus minimization?

IX. Experimental Setup

Both tools were evaluated on the state-of-the-art MAGMA data-set. MAGMA is a ground-truth fuzzing benchmark that enables uniform fuzzer evaluation and comparison [14]. The MAGMA data-set contains the fuzzers AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, honggfuzz, and SymCC-AFL aswell as a set of binaries with patched in bugs from libpng, libsndfile, libtiff, libxml2, lua, openssl, php, poppler and sqlite3.

For the evaluation of the trace clustering tool the binaries selected were libpng, libsndfile, libxml2 and sqlite3. This was due to their high range of differing bugs in the proof-of-concept dumps available for download from the MAGMA homepage, see Fig 5.

Fig. 5: Binary alongside bugs and crashes in sample used in clustering evaluation.

Binary	**Bugs Present**	**Crashes Sampled**
libpng	2	554
libsndfile	3	51
libxml2	4	940
sqlite3	6	317

The clustering tool was evaluated by cluster purity, number of bugs in final clusters and number of clusters.

For the evaluation of the corpus minimization tool the binaries selected were libpng and libsndfile. This was due to the availability of image and sound files to test on, see Fig 6. The fuzzer used to evaluate code coverage before and after corpus minimization was AFL. Each campaign was run for four hours against each binary twice, once with no corpus minimization and once with corpus minimisation. The code coverage data was saved for each run to evaluate the time-to-coverage.

Fig. 6: Binary alongside initial corpus size used in corpus minimization evaluation.

Binary	**Initial Corpus Size**
libpng	67
libsndfile	75

X. Results

During the trace processing step bugs were lost for libsndfile and sqlite3, see Fig. 7, Therefore, it is reasonable to conclude that the trace processing step is not effective at maintaining accuracy of the data before clustering. It is vital that root cause clustering tools do not miss bugs that could be fixed by the tester.

Fig. 7: Binary alongside bugs and crashes present after trace processing.

Binary	**Bugs Present**	**Trace Count**
libpng	2	272
libsndfile	2	45
libxml2	4	896
sqlite3	5	306

It is evident from Fig. 8 that the effectiveness of root cause clustering highly depended on the binary under test. For example, the bugs present in the libsndfile binary were CVE-2011-2696 (integer overflow) and a heap write error inserted by the MAGMA developers. As the integer overflow usually causes a crash in a singular location as well as the heap write error only causing a crash at the location where the corrupted heap is accessed these bugs only manifest at unique crashing locations meaning they can be easily clustered based on the edge counts present in their control flow graphs.

Contrary to this the clustering tool was ineffective on the bugs sampled for the sqlite3 binary. The main bugs in the sqlite3 binary manifested at a range of differing crashing locations so their control flow graph edge counts did not relate meaningfully to the root cause.

From this it reasonable to conclude that for the crashing inputs to be effectively clustered by root cause the manifestations of each bug need to be in singular locations, hence this tool would not be effective in a practical setting.

Fig. 8: Binary alongside cluster count and average cluster purity produced by the hit count root cause clustering tool.

Binary	**Cluster Count**	**Average Purity**
libpng	10	85%
libsndfile	3	100%
libxml2	2	69.73%
sqlite3	6	56.54%

The corpus minimization tool managed to effectively minimize the number of initial inputs for the libpng and libsndfile binaries, see Fig. 9. It is important that these are decreased such that the fuzzer has to mutate the least number of inputs inputs during the initialisation phase.

Fig. 9: Binary alongside original corpus size and minimized corpus size.

Binary	**Original**	**Minimized**
libpng	67	56
libsndfile	75	47

It can be seen from Fig. 10 and Fig. 11 that the coverage ended higher for both the minimized runs and that for the libsndfile binary the coverage was higher for a majority of the run.

Fig. 10: Code coverage of libpng for two 4 hour AFL fuzzing campaigns with an original and minimized input corpus.

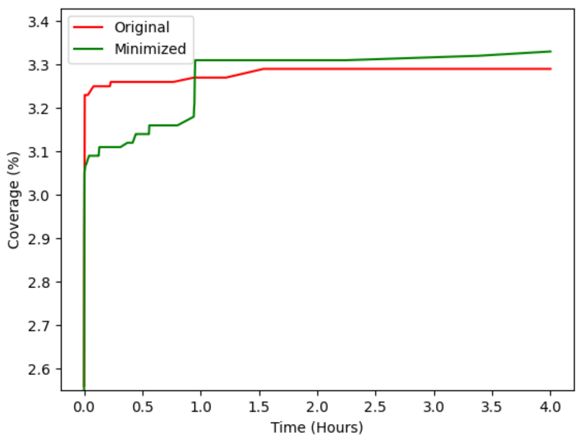
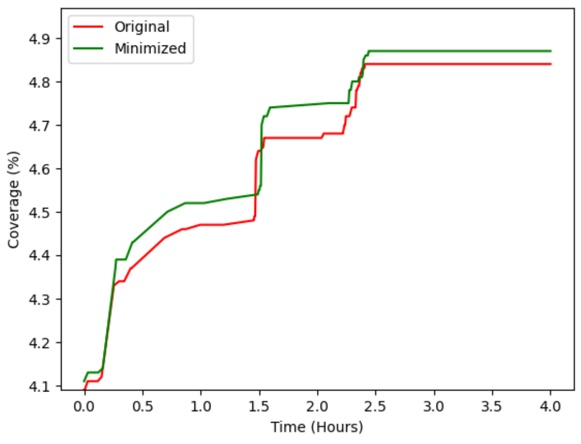


Fig. 11: Code coverage of libsndfile for two 4 hour AFL fuzzing campaigns with an original and minimized input corpus.



From this it may be concluded that the trace simplification and deduplication can be used for reasonably effective corpus minimisation.

XI. Conclusion

Fuzzing is an effective form of software testing that uses input mutation and generation to discover inputs resulting in crashes or hangs in software. Some limitations of fuzzing were highlighted as well as motivating factors for increased development in the field. Some projects aiming to overcome the issues of crash duplication such as Igor and AURORA were reviewed. Both of these had unique and effective approaches towards crash deduplication and root cause clustering. A tool, Moonlight, used for corpus distillation was also reviewed. These reviews showed that there are many areas in the field of fuzzing that could be improved upon.

The process of root cause clustering through trace simplification and edge count clustering was presented and evaluated. The clustering tool although effective on some binaries was found to be impractical if used in a realistic development setting.

A tool that used the trace simplification and then deduplcation for corpus minimization was also presented and evaluated. This tool showed somewhat promising results increasing time-to-coverage on the evaluated binaries for two 4 hour AFL fuzzing campaigns.

XII. GitHub

The repository containing all tools developed and documentation on how to use them can be found here https://github.com/nikolastyllis/Trace-Clustering-to-Aid-Fuzzing.git.

References

[1] G. Evron and N. Rathaus, "Fuzzing—what's that?" *Open Source Fuzzing Tools*, p. 11–26, 2007.

[2] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, p. 32–44, 1990.

[3] P. Godefroid, "Random testing for security: Blackbox vs. whitebox fuzzing," in *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, ser. RT '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 1. [Online]. Available: https://doi.org/10.1145/1292414.1292416

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[5] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[6] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[7] R. van Tonder, J. Kotheimer, and C. le Goues, "Semantic crash bucketing," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 612–622.

[8] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "Retracer: Triaging crashes by reverse execution from partial memory dumps," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 820–831.

[9] I. V. Krsul, "Software vulnerability analysis," Ph.D. dissertation, 1998, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2022-01-06. [Online]. Available: http://proxy.library.adelaide.edu.au/login?url=https://www.proquest.com/dissertations-theses/software-vulnerability-analysis/docview/304449527/se-2

[10] W. Liu, "Research on dos attack and detection programming," in *2009 Third International Symposium on Intelligent Information Technology Application*, vol. 1, 2009, pp. 207–210.

[11] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer, "Igor: Crash deduplication through root-cause clustering," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3318–3336. [Online]. Available: https://doi.org/10.1145/3460120.3485364

[12] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, "AURORA: Statistical crash analysis for automated root cause explanation," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 235–252. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko

[13] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using Context-Sensitive software fault injection," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2595–2612. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/jiang

[14] A. Hazimeh, A. Herrera, and M. Payer, “Magma: A ground-truth fuzzing benchmark,” *CoRR*, vol. abs/2009.01120, 2020. [Online]. Available: https://arxiv.org/abs/2009.01120

[15] A. Likas, N. Vlassis, and J. J. Verbeek, “The global k-means clustering algorithm,” *Pattern recognition*, vol. 36, no. 2, pp. 451–461, 2003.

[16] D. T. Pham, S. S. Dimov, and C. D. Nguyen, “Selection of k in k-means clustering,” *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 219, no. 1, pp. 103–119, 2005.