KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

The best master's thesis ever

ing. Ruben Kindt

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Software engineering

Promotor:
Prof. dr. Tias Guns

Academiejaar 2021 – 2022

Preface

I would like to thank everybody who kept me busy the last year, especially my promoter and my assistants. I would also like to thank the jury for reading the text. My sincere gratitude also goes to my wive and the rest of my family.

ing. Ruben Kindt

replace template by real one

Todo list

Contents

Abstract

abstract

The `abstract` environment contains a more extensive overview of the work. But it should be limited to one page.

Samenvatting

samenvatting

In dit `abstract` environment wordt een al dan niet uitgebreide Nederlandse samenvatting van het werk gegeven. Wanneer de tekst voor een Nederlandstalige master in het Engels wordt geschreven, wordt hier normaal een uitgebreide samenvatting verwacht, bijvoorbeeld een tiental bladzijden.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

CI/CD	Continuous Integration and Continuous Deployment, a pipeline for newly written code.to repeatably be: build, test, release, deploy and more.
CP	Constrain Programming Language sometimes also referred to as CPL
CPMpy	
PUT	Program Under Test, the piece of code, application of program we are testing on for potential bugs.
LLVM	Although it looks like an abbreviations, it is not. LLVM is the name of a project focused on compiler and toolchain technologies.
MUS	minimal unsat subset =/=
SMT	Satisfiability Modulo Theory

Symbols

$\neg$	negation
$\wedge$	logical and
$\vee$	logical or

Chapter 1

Introduction

There are a lot of causes for bugs: software complexity, multiple people writing different parts, changing objective goals, misaligned assumptions and more. Most these things can not be avoided during the creation of software but are the cause of program crashes, vulnerabilities or wrong outcomes. Multiple forms of prevention have been created like: the various forms of software testing, documentation, automatic tests and code reviews. All with the aim to prevent the occurrence of bugs and to reduce the cost associated with them. While automatic test cases often evaluate the goals of software end evaluate previous known bugs, it can do much more. Fuzzing software is a part of those automatic tests, a technique that is popular in the security world for exploit prevention. This technique generates random input for a program under test (PUT) and monitors if the program crashes or not. This explanation was the original interpretation of fuzzing as preformed by Miller[13], today this technique is seen as random generation based black box fuzzing while the current fuzzing envelops a broader term, as Manès et al.[11] put it nicely:

> "Fuzzing refers to a process of repeatedly running a program with generated inputs that may be syntactically or semantically malformed."

as quoted from [11]. With this technique we will try to detect bugs in the constraint programming and modeling library CPMpy [7] created by Prof. dr. Guns et al.

modus operandi bij intro

1.1 The usage of fuzzers in the software development cycle

During the development phase of software, tests are preformed to check if the written code matches the expected and wanted output. This can be done by the developers themselves or by quality assurance testers which do this full time and this on multiple different ways: code review, manual testing or automated testing. Those could exist out of unit tests, checking for known bugs, confirming that the use cases are working, code audits, fuzzing and others. None of the techniques mentioned above can prevent all possible bugs from occurring on top of that using only a single technique would cost more to find the same level of bugs then using multiple. While fuzzing emerged in

the academic literature at the start of the nineties, its full adoption thirty years later in the industry is still ongoing. With multiple companies like Google, Microsoft and LLVM creating their own fuzzers together with a pushing security sector the adoption has started to become a part of the growing toolchain for software verification.

1.2 Fuzzing and security

The adoption of fuzzers has definitely gained speed due to its proven effectiveness in finding security exploits. For example ShellShock, Heartbleed, Log4Shell, Foreshadow and KRACK could have been found using fuzz testing as shown in multiple sources [18], [2], [22], [9] and fuzzing is even recommended by the authors to prevent similar exploits [21] and [20].

1.3 Constraint programming in general

1.4 CPMpy

1.5 fuzzing history

Chapter 2

Fuzzing

The rise of fuzzing came when Miller gave a classroom assignment[15] in 1988 to his computer science students to test Unix utilities with randomly generated inputs with the goal to break the utilities. Two years later in December he wrote a paper[13] about the remarkable results. That more than 24% to 33% of the programs tested crashed. In the last thirty years the technique of fuzzing has changed significantly and various innovations have come forward. In this chapter we will look at classifications made, what the fuzzer expects as input, what we can get as output and we will look at the most popular fuzzers. The three[10][11] most used classifications are: how does the fuzzer create input, how well is the input structured and does the fuzzer have knowledge of the program under test (PUT)?

2.1 Generation and mutation

A fuzzer can construct input for a PUT in two ways, it can generate input itself or it can take an existing input and modify it, those original inputs are often called seeds. While Generation is more common than modification when it comes to in smaller inputs the opposite is true for larger inputs. This is cause by the fact that generating semi-valid input becomes a lot harder the longer the input becomes. For example, generating the word "Fuzzing" by uniformly random sampling ASCII, has a chance of one in $5 * 10^{14}$ of happening, making this technique infeasible when we want to generate bigger semi-valid inputs. With mutation we can start of with larger and already valid input and make modifications to create semi-valid inputs. With this last technique the diversity of the seeding inputs does become quite important. Ideally we would have an unlimited diverse set of inputs, but due to limited computation and available inputs we sometimes need to take a subset. In a paper by Alexandre Rebert et al. [19] they propose that seed selection algorithms can improve results and compare random seed selection to the minimal subset of seeds with the highest code coverage among other algorithms.

2.2 Input structure

While we have discussed the bigger scope on how inputs are created, let us go into more detail. As we have seen before fuzzing started mainly with Miller's classroom assignment, this random generation of inputs falls under 'dumb' fuzzing. Due to only seeing the input as one long list of strings with no knowledge of any sub-strings. This technique can be applied to mutational fuzzing as well. Compared to only adding symbols with generational fuzzing here we also remove or change randomly selected symbols. We can create three types of inputs: valid, semi-valid and nonsense input. With nonsense input we will almost be exclusively testing the syntactic stage of the PUT, often called the parser. Either the input crashes the parser or the parser will return invalid and the PUT will stop running. With semi-valid input we hope to be as close as possible to valid input to be able to explore the PUT beyond the parser and to catch an edge cases here. A smarter technique is referred to one which has knowledge about the structure inputs can have or should have. This increases the chance of inputs passing the parser, being able to test the deeper parts of the PUT and as such covering more of the PUT's code. At the cost of needing a more complex fuzzer. We can build a 'smart' fuzzer by adding knowledge about keywords, making it a lexical fuzzer, adding knowledge about syntax, for a syntactical fuzzer which can for example match all parentheses. Directed fuzz testing does fit in this category as well but it is not possible in a black box environment, more on that later.

2.3 Black, gray and white box fuzzing

Now that we have discussed adding knowledge of inputs to the fuzzer, we can also add knowledge about the PUT to the fuzzer. Which brings us to black, gray and white box fuzzing. With black box fuzzing we have no knowledge about the inner working of the PUT and we treat the PUT as a literal black box, we present out input and we look at what comes out of it. And with this minimal information the fuzzer then tries to improve its input creation. Compared to black box fuzzing gray box fuzzing usually comes with tools that give indirect information to the fuzzer, tools like: code coverage measurements, timings, types of errors and more. And as you may have predict white box testing is the term used when the fuzzer as much as possible. It will have access to the source code and can adjust their inputs to fuzz specific parts of the code. This at a higher cost due to having to reverse engineer the path to specific edge cases, meaning that white box fuzzing can find more bugs per input but creating those inputs take more time compared to black box fuzzing. The differentiation between black, gray and white box fuzzing is not clear cut, most people would agree that white box fuzzing has full knowledge about the PUT, including the source code, that gray box fuzzing has some knowledge about the PUT and that black box fuzzing has little to no knowledge about the PUT. Going into more detail all we can say is that it is no longer a black-and-white situation and that the lines become fuzzy.

is this pun allowed?

2.4 Fuzzer classification

Now that we know how we can classify fuzzers let us apply this to existing fuzzers to see how they work. For starters Miller's original work, which we discussed earlier, was random generation based black box fuzzing. His later work in 1995 on more UNIX utilities and X-Windows servers[14], his work in 2000 on Windows NT 4.0 and Windows 2000[6] and his work on MacOS [16] all fall in the same category of random generation based black box fuzzing. A couple of years later, KLEE was developed[4] by Cadar et al. KLEE is a generation based white box fuzzing tool build with the idea that bugs could be on any code path and the fuzzer generates inputs from the feedback it got from the symbolic processor and the interpreter this to increase the code coverage. A code coverage tool checks what lines of code are executed during the testing phase, with a higher percentage meaning that we used new lines of code. Using a line of code does not mean that the line of code has been found to contain no bugs, but not passing lines of code definitely means that the lines are untested. With the highest use case being the checking a specific test raises the code coverage, meaning that test uses a part of the code base that has not been tested yet. This together with the fact that getting a high code coverage is demanding task so that you don't max the metric out most of the time turns code coverage into a well rounded measurement. Among the more popular ones is the American fuzzy lop[1] (AFL), named after a rabbit breed, is a C and C++ focused, mutation based, gray box fuzzer released by Google but due inactivity the fork AFL++[5][2] has become more popular than the original and is maintained actively by the community. Not only did AFL spark AFL++, it has also sparked a python focused version pythonAFL[3], a Ruby[4] focused one, a Go[5] focused version and is shown by Robert Heaton[8] to not be difficult to write a wrapper for it. A potential reason to the inactivity of Google on the ALF project could be the development of both Clusterfuzz[6] and OSS-fuzz[7], a scalable fuzzing infrastructure and a combination of multiple fuzzers respectively. With the former one being used in OSS-fuzz as a back end to create a distributed execution environment. This with quite a bit of success

> "As of July 2022, OSS-Fuzz has found over 40,500 bugs in 650 open source projects."

update this in September

according to the repository itself. Not only Google has come with a fuzzer, Microsoft has jumped on board of fuzzing with OneFuzz[8] a self-hosted Fuzzing-As-A-Service platform which is intended to be integrated with the CI/CD pipeline. Although

[1] https://github.com/google/AFL
[2] https://github.com/AFLplusplus/AFLplusplus
[3] https://github.com/jwilk/python-afl
[4] https://github.com/richo/afl-ruby
[5] https://github.com/aflgo/aflgo
[6] https://google.github.io/clusterfuzz/
[7] https://google.github.io/oss-fuzz/
[8] https://github.com/microsoft/onefuzz

looking at the given stars on the Github repository, it looks like Google's tools are more popular than Microsoft' ones. A last prominent fuzzer we are going to take a small look at is the Libfuzzer[9] made by LLVM, a generation based, gray box fuzzer which is a part of the bigger LLVM project[10] with the focus on the C ecosystem. Being in the same ecosystem as AFl, LibFuzzer can be used together with AFL and even share the same seed inputs, sometimes called a corpus.

2.4.1 Types of bugs

Depending on what the output is of the fuzzer we can classify the types of bugs, as done in a recent paper[12] by Mansur: crashes, wrongly satisfied, wrongly unsatisfied or hanging. With some of these bugs being less acceptable then others For example, as a recent paper[12] by Mansur et al. describes, a crash for a constraint programming language (CP) is preferred over a wrongly unsatisfied model, since there is no way for the user to know that the solver failed (except for differentiation testing, more on that later). Meaning that the user will treat the result (wrongly) as correct compare this to a crash were it is clear that something went wrong. With hanging PUT's the user can not draw incorrect conclusions and with wrongly satisfied models the user can check the model's instances and evaluate the result before using it further. This is due to the fact that problems are frequently np-hard meaning they are easy to confirm but hard to solve. For practical reasons we will later change the undecidable and or hanging PUT's into timeouts. We know that the types of bugs can be classified in more detail, for example crashes into buffer overflows, invalid memory addressing and so on, but we choose to stay with a more general overview for now. An interesting classification to be added is the knowledge whether or not the bug is in the parser or not, as the authors of "Semantic Fuzzing with Zest"[17] would classify, is the bug in the syntactical or in the semantical part of the program?

2.5 The oracle problem

The oracle problem describes the issue of telling if a PUT's output was, given the input, correct or not or as said in "The Oracle Problem in Software Testing: A Survey"[1]

> "Given an input for a system, the challenge of distinguishing the corresponding desired, correct behavior from potentially incorrect behavior is called the test oracle problem."

by Barr et al. In their paper they discuss four categories: specified test oracles, derived test oracles, implicit test oracles, and the absence of test oracle. The biggest category would be the specified test oracles which contains all the possible encoding of specifications like modeling languages UML, Event-B and more. Their derived test oracles classification contains all forms of knowledge obtained from documentation

[9] https://llvm.org/docs/LibFuzzer.html
[10] https://github.com/llvm/llvm-project/

on how the program should work or previous versions of the program. The last two oracles categories come down to the use of knowing that crashes are always unwanted and the human oracle like crowdsourcing respectively.

2.5.1 Handling the oracle problem

Although the approach of by Bugariu and Müller in "Automatically testing string solvers"[3] falls in the first category mentioned above, their approach is innovative. While most fuzzers either use crashes or differential testing (more on that later) to find bugs, they know the (un)satisfiability of their formulas by the way of they are constructed. For satisfiable formulas they generate trivial formulas and then by satisfiability preserving transformations increase the complexity and for unsatisfiable formulas they use ¬ A ∧ A', with A' being a equivalent formula of A, to create the trivial unsatisfiable formulas. To increase the complexity of those trivial formulas, they again depend on satisfiability preserving transformation. This technique has also been applied to SMT solvers by Mansur et al. called STORM[12] this with mutational input creation compared to the previous generation based technique. In the paper the authors dissect all assertions into their sub-formulas and create an initial pool. In this pool the sub-formulas are checked if they satisfy or not and with this knowledge new formulas are created for the population pool with ground truth.

2.5.2 Differential testing

As mentioned above most fuzzers use either crashes to detect that the PUT has failed to provide a correct output or in cases where possible use differential testing. This last one uses a single or multiple analogue programs to test if the PUt gave the same output as the analogue programs. neither techniques is complete: crash based fuzzing can not detect wrong outputs and differential testing can not catch bugs that also occur in the analogue programs.

2.6 Conclusion

The final section of the chapter gives an overview of the important results of this chapter. This implies that the introductory chapter and the concluding chapter don't need a conclusion.

Chapter 3

Simplifying crashes

3.1 The First Topic of this Chapter

3.2 Conclusion

The final section of the chapter gives an overview of the important results of this chapter. This implies that the introductory chapter and the concluding chapter don't need a conclusion.

Chapter 4

CCPMpy

intro ch 4

4.1 The First Topic of this Chapter

4.2 Conclusion

The final section of the chapter gives an overview of the important results of this chapter. This implies that the introductory chapter and the concluding chapter don't need a conclusion.

Chapter 5

Evaluation

intro ch 5

5.1 The First Topic of this Chapter

5.2 Conclusion

The final section of the chapter gives an overview of the important results of this chapter. This implies that the introductory chapter and the concluding chapter don't need a conclusion.

Chapter 6

The Final Chapter

6.1 The First Topic of this Chapter

6.2 Conclusion

Chapter 7

Conclusion

The final chapter contains the overall conclusion. It also contains suggestions for future work and industrial applications.

Appendices

Appendix A

The First Appendix

Appendices hold useful data which is not essential to understand the work done in the master's thesis. An example is a (program) source. An appendix can also have sections as well as figures and references.

A.1 Lorem 51

Appendix B

The Last Appendix

Appendices are numbered with letters, but the sections and subsections use arabic numerals, as can be seen below.

B.1 Lorem 20-24

Bibliography

[1] Earl T Barr et al. "The oracle problem in software testing: A survey". In: *IEEE transactions on software engineering* 41.5 (2014), pp. 507–525.

[2] Hanno Böck. *How Heartbleed could've been found. Hanno's blog*. English. URL: https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html. 07/04/2015.

[3] Alexandra Bugariu and Peter Müller. "Automatically testing string solvers". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE. 2020, pp. 1459–1470.

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[5] Andrea Fioraldi et al. "AFL++ : Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[6] Justin Forrester and Barton Miller. "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing". In: *4th USENIX Windows Systems Symposium (4th USENIX Windows Systems Symposium)*. Seattle, WA: USENIX Association, Aug. 2000. URL: https://www.usenix.org/conference/4th-usenix-windows-systems-symposium/empirical-study-robustness-windows-nt-applications.

[7] Tias Guns. "Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example". In: *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*. Vol. 19. 2019. URL: https://github.com/CPMpy/cpmpy.

[8] Robbert Heatson. *How to write an afl wrapper for any language*. English. URL: https://robertheaton.com/2019/07/08/how-to-write-an-afl-wrapper-for-any-language/. 07/08/2019.

[9] Jaewon Hur et al. "Difuzzrtl: Differential fuzz testing to find cpu bugs". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1286–1303.

[10] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1.1 (2018), pp. 1–13.

[11] Valentin JM Manès et al. “The art, science, and engineering of fuzzing: A survey”. In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2312–2331.

[12] Muhammad Numair Mansur et al. “Detecting critical bugs in SMT solvers using blackbox mutational fuzzing”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 701–712.

[13] Barton P Miller, Louis Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[14] Barton P Miller et al. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1995.

[15] Barton P. Miller. *Fall 1988 CS736 Project List*. English. Project List. Computer Sciences Department, University of Wisconsin-Madison. URL: http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf.

[16] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. “An Empirical Study of the Robustness of MacOS Applications Using Random Testing”. In: *Proceedings of the 1st International Workshop on Random Testing*. RT ’06. Portland, Maine: Association for Computing Machinery, 2006, pp. 46–54. ISBN: 159593457X. DOI: 10.1145/1145735.1145743. URL: https://doi-org.kuleuven.e-bronnen.be/10.1145/1145735.1145743.

[17] Rohan Padhye et al. “Semantic fuzzing with zest”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 329–340.

[18] Pierluigi Paganini. *Exploiting and verifying shellshock: CVE-2014-6271. The Bash Bug vulnerability (CVE-2014-6271)*. English. URL: https://resources.infosecinstitute.com/topic/bash-bug-cve-2014-6271-critical-vulnerability-scaring-internet/. 27/09/2014.

[19] Alexandre Rebert et al. “Optimizing seed selection for fuzzing”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 861–875.

[20] Jo Van Bulck. “Microarchitectural Side-channel Attacks for Privileged Software Adversaries”. In: (2020).

[21] Mathy Vanhoef and Frank Piessens. “Release the Kraken: new KRACKs in the 802.11 Standard”. In: *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[22] Patrick Ventuzelo. *Can we find Log4Shell with Java Fuzzing? (CVE-2021-44228 - Log4j RCE)*. English. fuzzinglabs. URL: https://fuzzinglabs.com/log4shell-java-fuzzing-log4j-rce/. 13/12/2021.