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

Begeleider:
Ir. Ignace Bleukx

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.

replace template by real one

ing. Ruben Kindt

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[18], today this technique is seen as random generation based black box fuzzing while the current fuzzing envelops a broader term, as Manès et al.[15] 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 [15]. With this technique we will try to detect bugs in the constraint programming and modeling library CPMpy [11] 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, dynamic testing, 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 techniques. Sometimes a code audit is better, for example in situations where you want to know something easy that is most likely plainly written in the code. Other cases dynamic testing may be better, image you have a program which parses curricula vitae to check if candidates match the job position and you want to check if fresh Computer science graduates match the position software analyst. In this case it may be a lot easier to simulate the use case than to dive into the code. In situations where you want to test if bugs exist, you may not know where to start inside of the PUT, this is where Fuzzing may be the correct tool to use. Fuzzing emerged in the academic literature at the start of the nineties, while the industry's full adoption thirty years later is still ongoing. Multiple companies like Google, Microsoft and LLVM have created their own fuzzers and this together with a pushing security sector for the adoption has caused fuzzing 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 [23], [2], [27], [13] and fuzzing is even recommended by the authors to prevent similar exploits [26] and [25].

1.3 Constraint programming in general

1.4 CPMpy

1.5 fuzzing history

Chapter 2

Fuzzing

The rise of fuzzing came with Miller giving a classroom assignment[20] 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[18] 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 expect as output and we will look at the most popular fuzzers. The three most used classifications are[14][15][10]: 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 inputs for a PUT in two ways, it can generate input itself or it can take an existing input, called seeds, and modify them. While Generation is more common when it comes to smaller inputs, the opposite is true for larger inputs where modification has the upper hand. 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 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. [24] 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 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 independent symbols with no knowledge of any structure. This technique can be applied similarly 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: non-valid semi-valid and valid inputs. With non-valid inputs we will almost be exclusively testing the syntactic stage of the PUT, often called the parser. Either the input crashes the parser or it will be detected as invalid by the parser and the PUT will stop running. With semi-valid inputs we hope to be as close as possible to valid inputs in order to explore beyond the parser and to catch bugs deeper in the PUT. And lastly with valid input we are testing if the PUT behaves as expected and does not crashes. A smarter technique is referred to techniques, which have knowledge about the structure inputs can or should have. This increases the chance of inputs passing the parser and being able to test the deeper parts of the PUT, this at the cost of needing an increased complex fuzzer. We can build a 'smart' fuzzer by adding knowledge about keywords (making it a lexical fuzzer) or by adding knowledge about syntax (for a syntactical fuzzer, which can for example match all parentheses). Directed fuzz testing, where we guide the fuzzer on a specific path, does fit in this category of a 'smart' fuzzer as well but it is not possible in a black box environment, more on that in the next section.

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 provide input and we look at what comes out. 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, timings, classes of errors as measurements are all used as feedback, but more measurements are possible. Lastly, as you may have predicted, white box testing is the term used when the fuzzer has as much information to it available as possible. It will have access to the source code and can adjust their inputs to fuzz specific parts of the code (this falls under directed fuzzing). White box fuzzing does have a higher computation cost due to having to reverse engineer the path to specific edge cases, meaning that it can find more bugs per input but creating those inputs takes 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 has become fuzzy.

Ask permission to do this

2.4 Fuzzer classification

fix title?

Now that we know how we can classify fuzzers let us look at some 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[19], his work in 2000 on Windows NT 4.0 and Windows 2000[9], his work on MacOS[21] and his later revisit[17] all fall in the same category of random generation based black box fuzzing. A couple of years later, KLEE[5] was developed 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 that it should cover as much as possible. A code coverage tool is used to test which lines of code are executed and this together with the feedback it got from the symbolic processor and the interpreter the fuzzer can generate improved inputs. With this stride to obtain 100% code coverage it should be noted that covering a line of code does not mean that line of code has been found to contain no bugs, but not going over lines of code definitely means that the lines are untested. Therefore code coverage code coverage is sometimes used as a relative metric, checking if a specific test raises the code coverage, means that a test uses a new part of the code base that has not been tested yet. This combined with the fact that getting a high code coverage is a demanding task and does not easily gets max out turns code coverage into a well rounded measurement.

Among the more popular fuzzers, is the American fuzzy lop[1] (AFL), which named after a rabbit breed and is a C and C++ focused mutation based gray box fuzzer released by Google. But due inactivity on Google's part the fork AFL++[8][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[12] 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[7],

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

September update this

[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/

> source projects."

, 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 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[16] 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[16] 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"[22] 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."

[8] https://github.com/microsoft/onefuzz

[9] https://llvm.org/docs/LibFuzzer.html

[10] https://github.com/llvm/llvm-project/

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 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"[4] 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 of creating formulas satisfiable by construction has also been applied to SMT solvers by Mansur et al. called STORM[16] 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

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

Detecting crucial parts in inputs

algorithms?

paper 5 has nice simplification to isolation delta deb, called 'ddmin()' When we detect that the PUT crashes, wrongly satisfies, wrongly not satisfies or hangs on a given input we now want to know why it does that. What causes this unwanted output and what line the bug occurs. With crashes, a stack trace and some luck this could be easy, but when the crash is not main perpetrator or we get an other unwanted output the developer my need to debug deep into the code to find the bug. This with a potential large input could be a tedious and long assignment, for this reason we would like to know what parts of the input are related to the bug. We will discover this further in this chapter, starting with

plug in sections

3.1 Deobfuscating inputs

When receiving a big input the chance of having parts unrelated to the bug is almost guaranteed, we will call them (unintentionally) obfuscated inputs. Deobfuscating those takes a lot of try and error to see if the bug is still there[28] or having to walk through the execution to find the bug. Both take a while if we want to go to absolute minimal inputs, but for developers it is not needed to go to that extreme. As long as we take the bulk of the unrelated parts of inputs away it will help the developer to find the bug faster. With these techniques we can also group similar bugs and deduplicate error reports (more on that later) which is also fairly useful information for developers.

3.1.1 Simplifying

to find crucial parts of inputs is often done with simplification or Isolation, simplification is the technique where we remove parts of a failing input and check if it still fails and it often called "delta-debugging"[28] which belongs to the divide-and-conquer family of algorithms [3]. When it is no longer possible to remove any part of the input we have obtained an input where all parts are needed to expose the bug. This input is at the same time also the shortest possible input to trigger this bug, making finding the bug easier than in the original input filled with unrelated parts.

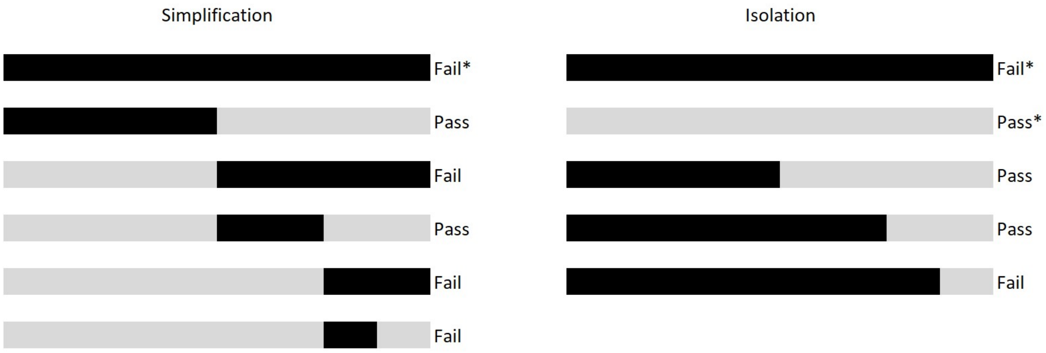

Figure 3.1: Deobfuscating inputs based on simplification (left) and isolation (right) on the same input. The '*' indicates that the result is already known and does not need to be recalculated. Figure based on an illustrations found in[28].

3.1.2 Isolation

Another technique, isolation, is explained by Andreas Zeller et al. [29] this is a technique where instead of minimizing the input we try to find the smallest difference between an input that shows the bug versus an input that does not show the bug. This with the advantage that no matter if we find the bug or not the difference will diminish, either the maximum input will shrink or the minimum input will grow. This technique brings extra complexity with the tracking of multiple inputs and the maximal input could take longer to run due to its size, but according to Andreas Zeller et al. this is the faster one to the two. Figure 3.1 shows the difference between simplifying and isolation, both finding the critical part of the input, with simplification the critical part is indicated by the last test in the figure while with isolation it it the differenc of the passed and failed test.

3.1.3 Alternative approach

An alternative approach by Alexandra Bugariu and Peter Müller[4] is to forgo the need of deobfuscating inputs by generating inputs "small by construction". Or by fuzzing in a similar way to when the bug was found as done by Muhammad Numair Mansur et al[16] and by adding a small limitation getting a smaller input.

3.2 The precision effect

With one caveat to take in mind we need to be careful to still find the same bug and to not change a null pointer dereference bug to a parser related bug. This, as discussed in the previous chapter, is due to some bugs being more important than others. In a paper by Andreas Zeller and Ralf Hildebrandt [29] talk about this exact problem which they called "the Precision Effect". Sometimes this is not a problem for example when we are trying to find all possible bugs and will rerun the fuzzer after each incremental improvement or the situation where a deeper bug turns into another deep bug. But overall we try to avoid this effect, which can be done with the techniques we will talk about in section 3.4.

3.3 What size to change

Another thing we glossed over is the chuck sizes to remove while trying to find the critical parts of the inputs. The previous seen techniques will work well on the original fuzz testing Miller et al.[18] worked on since those random generated symbols where independent from each other. When testing something more complex words like "while" or "float" we no longer can split on all possible places, since the input would most likely no longer parse. The same for splitting size, in the exact middle for figure 3.1 we conveniently took one-eighth of the input as the chuck sizes for the ease of the example. For performance reasons we hope we can keep our chuck sizes as big as possible to be able to discard larger unrelated parts of the inputs. but when this is not possible we will need to decrease the granularity of the chuck sizes. To be able to find the critical parts of an input of the form "XXooXooXXoo" (with 'o' being the critical parts and the 'X' being unrelated to the bug) we should always search further with same granularity while the removed parts are already removed until all options with that granularity searched[28]. This will make sure that we eliminate all unrelated parts with the specific granularity and get "ooXoooo".

We could also apply some techniques seen in section 2.2 where we discussed the creation of randomly and smarter created inputs. Instead of removing (hopefully) unrelated parts based purely on where the part sits in the input. We can use knowledge of the input structure or knowledge of the PUT to guide us in the removal[28], both lexical (the meaning of words) and syntactical knowledge (the meaning of combinations of words) can be used to help us in deobfuscating inputs. Where syntactical knowledge would help us remove the most since it is the bigger of the two.

3.3.1 Preserving satisfiability

With techniques as mentioned in section 2.5.1, satisfiable by construction will need to take in mind the extra complexity of preserving the ground truth when deobfuscating inputs. either we can apply Muhammad Numair Mansur et al.'s[16] technique of trying to fuzz the same bug again, but with less symbols or as Robert Brummayer and Armin Biere[3] did use other SMT solvers.

NEEDS example

3.4 Deduplication

Another thing to notice is that multiple inputs could prompt the same bug from occurring, these inputs could be similar but don't have to be. With simplifying the input we should be able to detect exact copies, but depending on the simplification's time complexity other techniques could be better with similar results. In case where we would have access to stack traces (via crashes or hanging PUT's) we could differentiate the bugs on basis of the hash of multiple lines from the backtrace sometimes even numerous hashes per input. this technique is called stack backtrace hashing and is quite popular according to Valentin J.M. Manès et al[15]. Another technique talked about in that paper, is looking at the code coverage generated by the inputs where we use the executed path (or hash of it) is used as a fingerprint of the inputs. A technique, used by Microsoft[6] is called semantics based deduplication, where in stead of back track use memory dumps to hopefully find the origins of bugs. This use of dumps is less ideal due to traces having more information, but the latter is not always possible due to the performance overhead and privacy causes as specified in the paper. A last technique would be looking at the bug description left by a manual bug reports by the user, although this dependence on the quality of the bug reports and is most likely poorly automatable. None of the techniques mentioned above are perfect: with stack backtrace hashing you could find to many false positives or false negatives depending on the depth taken from the stack, with coverage some inputs will generate extra function calls and the semantics based deduplication are limited to X86 or x86-64 code with the binary file and the debug information. Neither of these techniques work with black box fuzzing unfortunately.

3.5 Conclusion

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

CPMpy

intro ch 4

4.1 CPMpy

4.2 Innerworkings of CPMpy

4.2.1 First part

4.2.2 Second part

4.3 history of CP(Mpy)

4.3.1 Numberjack

ocverview of more CP's

4.3.2 Z3

4.4 Conclusion

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

The Final Chapter

5.1 Conclusion

Chapter 6

Conclusion

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

Appendices

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] Robert Brummayer and Armin Biere. "Fuzzing and delta-debugging SMT solvers". In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 2009, pp. 1–5.

[4] 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.

[5] 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.

[6] Weidong Cui et al. "Retracer: Triaging crashes by reverse execution from partial memory dumps". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 820–831.

[7] Zhen Yu Ding and Claire Le Goues. "An Empirical Study of OSS-Fuzz Bugs". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 131–142.

[8] 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.

[9] 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.

[10] Patrice Godefroid. "Fuzzing: Hack, art, and science". In: *Communications of the ACM* 63.2 (2020), pp. 70–76.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] Barton Miller, Mengxiao Zhang, and Elisa Heymann. “The relevance of classic fuzz testing: Have we solved this one?” In: *IEEE Transactions on Software Engineering* (2020), pp. 285–286.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[29] Andreas Zeller and Ralf Hildebrandt. “Simplifying and isolating failure-inducing input”. In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200.