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.

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

Listings

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
CPL	Constrain Programming Language also referred to as CP
CNF	Conjunctive Normal Form, which is a boolean formula written using conjunctions of distinctions.
CSP	Constraint Satisfaction Problem is a problem with constraints and variable with a specific domain e.g. finite.
CPMpy	Constraint Programming and Modeling language for PYthon.
CVC	Cooperating Validity Checker
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 Unsatisfiable Subset =/=
SMT	Satisfiability Modulo Theory
SUT	Sofware under Test

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[31], today this technique is seen as random generation based black box fuzzing while the current fuzzing envelops a broader term, as Manès et al.[25] 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 [25]. With this technique we will try to detect bugs in the constraint programming and modeling library CPMpy [18] 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

bit more

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 [7, 20, 38, 44] and fuzzing is even recommended by the authors to prevent similar exploits [42, 43].

1.3 Constraint solving in general

1.4 CPMpy

1.5 fuzzing history

Chapter 2

Fuzzing

The rise of fuzzing came with Miller giving a classroom assignment [33] 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 [31] 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.

2.1 Classifications

The three most popular classifications are [17, 23, 25]: how does the fuzzer create input, how well is the input structured and does the fuzzer have knowledge of the program under test (PUT)?

2.1.1 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. [39] 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.

2.1.2 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.

2.1.3 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.

2.2 Classifying popular fuzzers

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 a random generation based black box fuzzing. And started off as an assignment for his students to test the reliability of Unix utility programs by trying to break them using a fuzz generator, which was able to generate printable ASCII, non-printable ASCII, with or without null terminating characters of a random length. That resulted in a successful paper [31] two years later. His later work in 1995 on even more UNIX utilities, his work on X-Windows servers [32], his work in 2000 on Windows NT 4.0 and Windows 2000 [15], his work on MacOS [34] and his later revisit [30] on fuzzing all fall in the same category of random generation based black box fuzzing. This papers showed that a significant portion of programs are able to be crashed with random inputs. Of the programs tested 15 to 43% of the Unix utilities crashed, 6% of the open-source GNU utilities crashed, 26% of X-Window applications crashed, 45% of Windows NT 4.0 and Windows 2000 programs crashed and 16% MacOS programs crashed.

A couple of years later, KLEE [10] 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 testing should cover as code much as possible. A code coverage tool is used to test which lines of code are executed and this combined with the feedback KLEE got from the symbolic processor and the interpreter it can generate improved inputs. KLEE does this by symbolically executing the program executions branching on any dangerous operations and when it finds an error it will convert the symbolic to a concrete representation based on the constraints it needed to get passed the specific branches and uses this concrete representation to test the original program.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 remain 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 to 100% turns code coverage into a well rounded measurement.

As for 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

[1] `https://github.com/google/AFL`

released by Google. But due inactivity on Google's part the fork AFL++[2] has become more popular than the original and is maintained actively by the community [14]. Not only is it actively maintained, it is also actively used by researchers. Not only did AFL spark AFL++, it has also sparked a python[3] focused version, a Ruby[4] focused one, a Go[5] focused version and is shown by Robert Heaton [19] 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 [12],

September update this

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

according to the repository itself. Not only Google has come forward with a fuzzer. Even 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. The last prominent fuzzer we are going to take a 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.

2.2.1 Testing CP and SMT with Fuzzers

Until now we discussed fuzzers more generally, we would like to deliberate specific fuzzers build for testing constraint programming languages (CP) and satisfiability modulo theory (SMT) solvers.

One of those fuzzers is STORM which is a mutation based black box fuzzer created by Muhammad Numair Mansur et al. [26] to find critical bugs in SMT solvers. In their paper they explain the inner working thoroughly, but briefly summarized STORM creates an initial pool of smaller formulas from existing formulas found in seeds, uses another solver to create models of those smaller formulas. To then construct more complex formulas with the knowledge of their ground truth, with this STORM can test the SMT solver as can be seen in figure 2.1. This novel way of

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

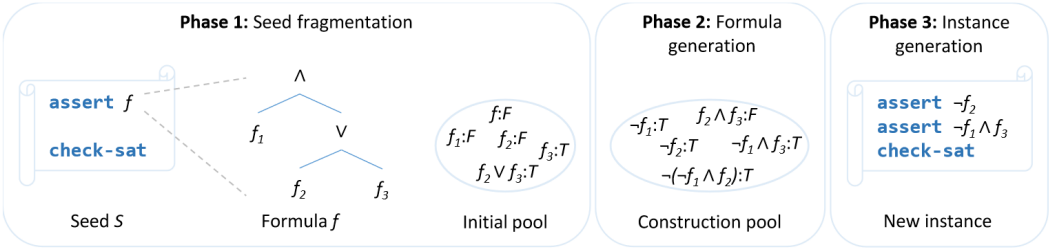

FIGURE 2.1: Overview of the three STORM phases as presented by Muhammad Numair Mansur et al. in "Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing" [26].

fuzzing SMT solvers with inputs that are satisfiable by construction and has been cited significantly considering that it is a recent paper.

Another technique for fuzzing SMT solvers is the one proposed by Dominik Winterer et al. with their fuzzer YinYang [46], which uses "Semantic Fusion" to test the solvers.

> "Our key idea is to fuse two existing equisatisfiable (i.e., both satisfiable or unsatisfiable) formulas into a new formula that combines the structures of its ancestors in a novel manner and preserves the satisfiability by construction. This fused formula is then used for validating SMT solvers."

As quoted from "Validating SMT Solvers via Semantic Fusion" [46] Dominik Winterer et al. take a free variable from each of the equisatisfiable formulas to be able to create a new variable using a reversible fusion function. For example a formula $\phi_1 =$ X $>$ 10, $\phi_2 =$ Y $<$ 9 with the fusion function for Z = X + Y would become $\phi_3 =$ Z - Y $>$ 10 $\wedge$ Z - X $<$ 9, linking both satisfiable formulas together. For unsatisfiable formulas an extra conjunction is needed with the definition of the new variable, because a substitution could result in the loss of the unsatisfiability of the formula as mentioned in the paper. The results of the paper where also significant with 45 bugs in state-of-the-art SMT solvers in Z3[11] and CVC4[12]. Dominik Winterer et al. also give multiple fusion functions like multiplication and string concatenations which can be applied to integers and real numbers and strings respectively. Extending this technique to other data types or more fusion functions would not be difficult.

A last fuzzer we are going to discuss is Falcon, a fuzzer that extends the search space to also test the configuration of the SMT solvers. This fuzzer made by Peisen Yao et al. [47] found quite the success by firstly linking the configuration options to the operations and to then use this information to fuzz better When using STORM as a fuzzer with the knowledge of the configuration space the authors managed to increase the code coverage by 17.2 to 18.8%. When knowing that SMT solver like Z3 and CVC4 contain more than 700 000 and 100 000 lines of code respectively means that any percentage is a significant amount of extra lines covered.

[11] `https://github.com/Z3Prover/z3`
[12] `https://github.com/CVC4/CVC4-archived`

2.3 Other forms of testing

2.3.1 Differential testing

As mentioned above a lot of fuzzers use crashes to detect that the PUT has failed to provide a correct output or when possible use differential testing. This latter one uses a single or multiple analogue programs to test if the PUT gave the same output as the analogue programs. As Christian Klinger et al. did in their paper [22]. Neither crash based nor differential testing is ideal: crash based fuzzing can not detect wrong outputs and differential testing requires that one or multiple analogue programs exits preferably with an different implementation to reduce overlapping bugs. The latter technique may therefore not always be possible due to the existence of those analogue programs.

2.3.2 Metamorphic testing

In situations where the existence of analogue programs would be limited metamorphic testing could be a solution. This technique uses knowledge of the domain to tell if subsequent solutions may be wrong. For example in "TestMC: Testing Model Counters using Differential and Metamorphic Testing" [41] the authors use add an extra restriction to a variable in a formula to test if the amount of models reduces (or remains equal). Or that the amount of models remain the same when giving a equivalent formula or adding a tautology compared to the original. This technique no longer depends on a secondary oracle but does depend on multiple executions and could miss a bug that occurs in multiple situations. This technique has been applied in combination with differential testing to test constraint solvers [1] by Akgün, Özgür et al. with success.

2.4 Types of bugs

bugs bad 1.2

Not only can we classify fuzzers, we can also classify the types of bugs found by the fuzzers, as done in a recent paper [26] by Muhammad Numair Mansur et al. being: crashes, wrongly satisfied, wrongly unsatisfied or a hanging PUT. With some of these bugs being less acceptable then others. For example, as Muhammad Numair Mansur et al. describes, a crash is preferred for a constraint programming language (CP) over a wrongly unsatisfied model, since there is no way for the user to know that the solver failed in that last case (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 confirm 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. To this extensive we will be adding a wrong model, unknown and for practical reasons move a hanging PUT to the unknown category using a timeout. We are aware that the types of bugs can be classified in even 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 part of the PUT or not. The put could already fail on inputs during the interpretation of the inputs and as discussed we would also like to detect bugs deeper in the PUT. As the authors of "Semantic Fuzzing with Zest" [37] 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" [4]

> "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 oracles. 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 by knowledge of 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" [9] 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 $\neg$ A $\wedge$ 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 Muhammad Numair Mansur et al. called STORM [26] which uses mutational input creation compared to the previous generation based techniques. In the paper the authors dissect all SMT 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, from this pool new theories are created and tested. This makes that STORM does not need an oracle to test the entire theory, but only the smaller sub-formulas.

2.6 Opinions against Fuzzing

We have talked about the successes of fuzzing, but there are also opinions against fuzzing. As William M. McKeeman [28] writes some developers do not like the automatic way of adding more bugs to their backlog and see it as unreasonable. "Why would a person do this (obscure actions)?" and that fuzzing seems to generate an infinite amount of bugs are also pet peeves of developers. Although we have a bias, due to writing a dissertation about fuzzing, but we think that those perspectives should not be left out this paper. But also mention that this is not a single view shared by all developers from [46, 47, 50] and more we see a positive response to newly found bugs. Some have even started implementing fuzzers [6] in their toolchain to find bugs.

2.7 Conclusion

In this chapter we have seen an overview of what fuzzers are, which are used in the industry and how they work. We have seen techniques and fuzzer specified for SMT solver but also for constraint solvers. To finally end with some problems around getting the difference between correct and incorrect behavior and a small paragraph on the perspective of the developers on fuzzing.

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.

does this chapter need a conclusion?

Chapter 3

Detecting crucial parts in inputs

When we detect that the PUT crashes, wrongly satisfies, wrongly unsatisfies, hangs or gives the wrong solution on a given input we want to know why it does that. What causes this unwanted output and on what line the bug occurs. With crashes, a stack trace and some luck this could be easy, but when a bug causes a crash in another place or we get another unwanted output the developer may need to debug deep into the code to find the bug. This with a potential large inputs created by the fuzzer 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 deobfuscating inputs.

3.1 Deobfuscating inputs

> "Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it." [1]
> -Richard Stallman, Using and Porting GNU CC

When receiving a big input the chance of it having parts unrelated to the bug is almost guaranteed, we will call these inputs (unintentionally) obfuscated inputs. Deobfuscating those inputs can take a lot of trying to see which variations still show the bugs or having to walk through the execution to find the bugs. 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 duplicate errors (more on that later) which is also fairly useful information for developers.

3.1.1 Simplifying

To find crucial parts of inputs, it is often achieved either with simplification or Isolation. Simplification is the technique where we repeatably remove parts of a failing input and check if it still fails and it often called "delta-debugging", which

Minimizing Delta Debugging Algorithm

Let *test* and $c_{\text{✘}}$ be given such that $test(\emptyset) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $c'_{\text{✘}} = ddmin(c_{\text{✘}})$ such that $c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✘}}) = \text{✘}$, and $c'_{\text{✘}}$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_{\text{✘}}) = ddmin_2(c_{\text{✘}}, 2) \quad \text{where}$$

$$ddmin_2(c'_{\text{✘}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \text{✘} \text{ (“reduce to subset”)} \\ ddmin_2\big(\nabla_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✘} \text{ (“reduce to complement”)} \\ ddmin_2\big(c'_{\text{✘}}, \min(|c'_{\text{✘}}|, 2n)\big) & \text{else if } n < |c'_{\text{✘}}| \text{ (“increase granularity”)} \\ c'_{\text{✘}} & \text{otherwise (“done”).} \end{cases}$$

where $\nabla_i = c'_{\text{✘}} - \Delta_i$, $c'_{\text{✘}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✘}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{\text{✘}}) = \text{✘} \wedge n \leq |c'_{\text{✘}}|$.

Figure 3.1: A minimizing delta-debugging algorithm as shown in [49].

belongs to the divide-and-conquer family of algorithms [8]. The algorithm can be seen in figure 3.1 with "$c_{\text{✘}}$" meaning the failing input to be deobfuscated, "✔" meaning that a test passed with the given input, "✘" failed with the given input, "Δ" and "∇" being a subset of the input and the complement of the former and "1-minimal" meaning that not a single character can can change without the input going from failing to passing. Firstly we start the algorithm with the input and a split (n) of two. If there is a subset that still fails on it own, then we continue with that subset else we look for a subset where the complement of the input still fails but where a subsets is missing from the input. In the case where we split the input in two parts this would be the same as te previous. In case we do not find any smaller subset to continue on, then we reduce de granularity of the split by two. To finally when it is no longer possible to remove any part of the input we have obtained an input where all parts are necessary to expose the bug. This input is at the same time also the shortest possible input to trigger this bug, making finding the bug for the developer easier than in the original input filled with unrelated parts. An example of delta-debugging to minimize input can be found in figure 3.2. In the first two steps no removal of any part nor complement was possible therefore we reduces the granularity, after which a removal of parts 3 and 4 was found possible. To then use some previous knowledge (lines 9, 10 and 11) with 2 new tests to remove parts 5 and 6). To then decrease the granularity again until we reach a minimal input.

3.1.2 Isolation

The second technique, isolation, 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 bigger inputs often take longer, but according to Andreas Zeller et al. [49] this is the faster one to the two techniques. Figure 3.3

Step	Test case									*test*	
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	?	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	?	⇒ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	?	Testing $\Delta_1, \ldots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	✔	
5	Δ_3	.	.	.	.	5	6	.	.	✔	
6	Δ_4	.	.	.	.	.	.	7	8	?	
7	∇_1	.	.	3	4	5	6	7	8	?	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	✘	⇒ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	?*	Testing $\Delta_1, \Delta_2, \Delta_3$
10	Δ_2	.	.	.	.	5	6	.	.	✔*	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	?*	
12	∇_1	.	.	.	.	5	6	7	8	?	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	✘	⇒ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	?*	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	?*	⇒ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	?	Testing $\Delta_1, \ldots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	✔	
18	Δ_3	.	.	.	.	.	.	7	.	?	
19	Δ_4	.	.	.	.	.	.	.	8	?	
20	∇_1	.	2	.	.	.	.	7	8	?	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	✘	⇒ Reduce to $c'_{\text{✘}} = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	?*	Testing $\Delta_1, \ldots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	?*	
24	Δ_3	.	.	.	.	.	.	.	8	?*	
25	∇_1	.	.	.	.	.	.	7	8	?	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	?	
27	∇_3	1	.	.	.	.	.	7	.	?	Done
Result		1	.	.	.	.	.	7	8		

FIGURE 3.2: A minimizing delta-debugging example as shown in [49] with an input that is deobfuscated with the ddmin() algorithm3.1.

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 is the difference of the last passed and last failed tests.

3.1.3 Connection with minimal unsatisfiable subset and maximally satisfiable subsets

For readers that are familiar with the SMT of constraint solving-world will have noticed that this techniques feels similar to a way of finding the minimal unsatisfiable subset (MUS), which it is in the case of a solver (wrongly) stating that an input is unsatisfiable. With MUS you try to find the smallest subset of formulas (or constraints) that will result in an unsatisfiable solution while with MSS you would be trying to find the biggest set of formulas (or constraints) that would result in a satisfiable solution. Both are an iterative process and can be applied in the simplification and the isolation process. But solving which combination of formulas results in the smallest of biggest subset is an computationally intensive progress.

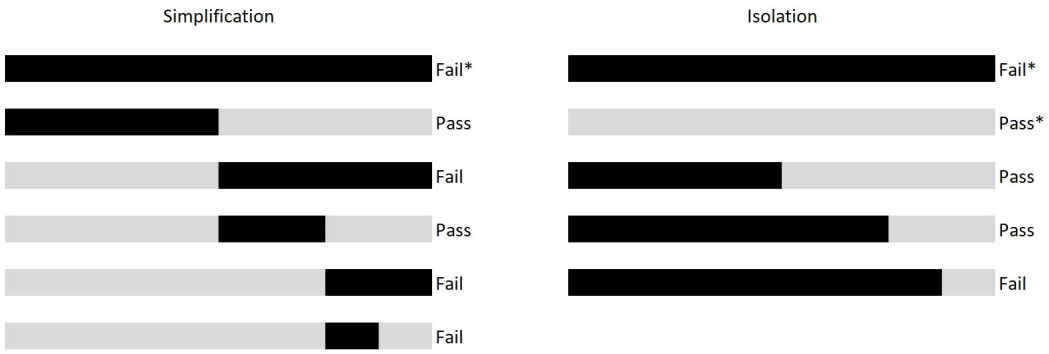

Figure 3.3: Deobfuscating inputs based on simplification (left) and isolation (right) on the same input. The '*' indicates that the result is already known and does not need to be recalculated. Figure based on an illustrations found in Why programs fail: a guide to systematic debugging by Andreas Zeller [48].

Fortunately a lot ot though has already been put in it to reduce it, for example Mark H. Liffiton et al. have proposed multiple "Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints" [24]. Again we should note that an Minimal input is not needed as we aim to reduces de input to help de developer find the error faster, a difference between a smaller and the absolute minimum will cause for a big difference in practice.

3.1.4 Alternative approach

An alternative approach compared to the already mentioned techniques is one by Alexandra Bugariu and Peter Müller [9] is to forgo the need of deobfuscating inputs by generating inputs "small by construction". Because the smaller the inputs are the space there is for remaining stuff to obfuscate the Input. On the other hand the chance of finding bigger bugs with multiple constraints interacting with each other will be harder to find. A last alternative approach would be retrying fuzzing by adding an increasing size limitation to find the same bug input as done by Muhammad Numair Mansur et al [26].

3.2 What size to change

A subject we glossed over is the chuck size, the size to remove while trying to find the critical parts of the inputs. The previous seen techniques will work well on the original fuzz testing by Miller et al. [31] since those random generated symbols where independent from each other. But when testing more complex words like function names we no longer can split on all possible places, since the input would most likely no longer parse. In figure 3.3 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. For example, 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 are searched [48]. This will make sure that we eliminate all unrelated parts with the specific granularity and get "ooXoooo" instead of "ooXooXXoo".

For more complex inputs we can apply techniques seen in section 2.1.2 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 [48]. Both lexical (the meaning of words) and syntactical knowledge (the meaning of combinations of words) can be used to help us in deobfuscating complex inputs. Where syntactical knowledge would help us remove the most since it is the bigger of the two.

3.2.1 Preserving satisfiability

With the techniques as mentioned in section 2.5.1, "satisfiable by construction" formed inputs will need to take the extra complexity of preserving the ground truth in mind when deobfuscating inputs. When the ground truth says that an input should be unsat and the PUT says it is a satisfiable problem with the following output, then we can not remove constraints to retest if that specific constraint was the cause without knowing the new ground truth. As potential change, this could switch the original input from an unsatisfiable to a satisfiable problem. We could use a trusted solver to make sure that we only retest unsat inputs by retesting each change. or as done by Muhammad Numair Mansur et al.'s [26] technique of trying to fuzz the same seed in the hope to find a smaller input that gives the same bug. Or use other SMT solvers to make sure that the ground truth does not change as Brummayer and Biere [8] did. In the other scenario when he ground truth says that an input should be sat with X amount of models and the PUT says that the input is unsatisfiable. Then we have more options to deobfuscate the inputs. We can use the previously mentioned techniques like the other scenario but we can also use simplifying, isolation, MUS, MSS and the technique of refuzzing like STORM while still preserving the (un)satisfiability of the problem.

3.3 The precision effect

This finding of the same bug needs to be done carefully, so that we do not change a null pointer dereference bug to a parser related bug. This, as discussed in the previous chapter, is because we value some bugs with more importance than others. In a paper by Andreas Zeller and Ralf Hildebrandt [49] they 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 in the following section.

3.4 Deduplication

With deobfuscating the inputs we can detect exact copies, but depending on the deobfuscation's time complexity other techniques could be better with similar results. In case where we would have access to stack traces (via crashes) we could differentiate the bugs on the basis of the hash from the backtrace, sometimes even numerous hashes per input depending on the amount of backtrace lines taken. This technique is called "stack backtrace hashing" and is quite popular according to Valentin J.M. Manès et al. [25] Another technique talked about in that paper, is looking at the code coverage generated by the inputs where the executed path (or hash of it) is used as a fingerprint of the inputs. A technique, used by Microsoft [11] is called "semantics based deduplication", where instead of backtrace they 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 manual bug reports, although this dependence on the quality of bug reports and is most likely poorly automatable. None of the techniques mentioned above are perfect: with stack backtrace hashing you need access to the backtrace, 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 those first techniques will 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

CP, SAT and SMT

intro ch CP and SMT

4.1 Holy grail of programming

> "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [16] quote by Eugene C. Freuder

As the quote and the paper from Eugene C. Freuder says, he and others believe that in the ideal world the user conveys any problem to the program and that the program will solve it. Which does match a coarse summary of what a constraint programming language (CP) can do. But we don't live in an ideal world and problems need to be converted or split up into a representation the solver can understand. This is where constraints come into the picture, constraints are mathematical or logical limitations; or connections between variables when solved form a model that satisfy all constraints and sometimes minimize or maximize a final objective function, for example finding a model where a postman visits all cities on a route but minimize the distance traveled. With CP the focus lays more on the solving high level problems with specialties around scheduling and planning [5]. Which their key feature being the global constraints, a set of "function" that are aimed at high level of abstraction. for example the "alldifferent()" which makes sure that all variables will receive a different value or "circuit()" that holds if the input forms a Hamiltonian circuit.

A second field of research that we will discuss is the boolean satisfiability problem better known as SAT, where the focus lays on the boolean variant of constraints within the family of constraint solvers. This field of research has produced quite a lot of progress due to its age, resulting in efficiency in solving [3].

Our last field of research that we will discuss, is the satisfiability modulo theory (SMT) which builds further on SAT. This by including Lists, arrays, strings, real numbers, integers and other more complex data structures and types. With SMT the focus lays more on static checking and program verification [3, 35].

4.2 Constraint programming

As mentioned before CP's are well versatile in the solving of constraints and especially when it comes to planning and scheduling. This by their efficient constraint propagation, backtracking and the linking of related constraints [45]. Originally CP's can be linked back to constraint logic programming (CLP) where programming languages (mostly logic programming languages) where combined with constraints. A notable version is that of Joxan Jaffart and Jean-Louis Lassez [21, 45] extension in Prolog.

To further introduces you to constraint programming we will show a popular puzzle in the CP field "send + more = money". which is a logic puzzle made by Henry Dudeney and published in Strand Magazine's July 1924 edition [13]. Where each character represents a single digit between zero and nine (both included), meaning that the 'e' from "send" should be the same value as the 'e' from "more" and "money". The other rules go as follows: all characters should have a different digit, the starting letter of each word can not be zero and after replacing all characters with digits the sum should be correct.

$$\begin{array}{r} \text{S E N D} \\ +\ \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$

This example is often used to introduce CP field, so will we. We will also use MiniZinc[1] as our constraint programming language to find the solution for this example.

```
1  include "globals.mzn";
2  var 1..9: S; % Since 'S' is a starting chacter it is limiter from 1 to 9
     included.
3  var 0..9: E; % Other charcaters are limited from 0 to 9 included.
4  var 0..9: N;
5  var 0..9: D;
6  var 1..9: M;
7  var 0..9: O;
8  var 0..9: R;
9  var 0..9: Y;
10
11 constraint % The sum must hold.
12               1000 * S + 100 * E + 10 * N + D
13             + 1000 * M + 100 * O + 10 * R + E
14 = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;
15
16 constraint alldifferent([S,E,N,D,M,O,R,Y]);
17
18 solve satisfy;
```

[1] https://www.minizinc.org/

```
19  output
20  [" \(S)\(E)\(N)\(D)\n",
21  "+ \(M)\(O)\(R)\(E)\n",
22  "= \(M)\(O)\(N)\(E)\(Y)\n"];
```

LISTING 4.1: Solution to "send + more = money" slightly modified from `https://www.minizinc.org/doc-2.5.5/en/downloads/send-more-money.mzn`

On line 1 you can see the importing of the global constraints, which CP is know for. Line 2 until 9 we can see the declarations of all possible characters to the possible digits, starting letters like 'S' and 'M' are limited by their representation from 1 to 9 instead of using a constraint. The constraint on line 11 runs through until line 14, where it is closed by a the semicolon. This constraint specifies the matching of the sum. At line 16 we use one of the imported functions namely "alldifferent()" which will satisfy if no similar values occur in the array. To then start solving for satisfiability at line 18, this in comparison of solving for minimization or maximization of an objectify function, which does not apply here. Finally after a solver has found satisfiability we print the result using a pretty print from line 19 to 22. We will leave it up to you for finding the solution of the puzzle, but remember that you can check your answer with the program above.

The advantage of using constraints is also visible by the sum constraint, if we had used allocations The "alldifferent" and the sum constraint also shows us the power of CP's, with a single statement multiple relations are expressed[27]. Instead of needing to specify all possible relations of the last tree characters (D + E = Y, E = Y - D and D = Y - E) a single expression suffices. When knowing two values we can infer the third, imagine the work needed for the "alldifferent()" constraint.

4.2.1 origin

Within constraint programming we can distinguish two branches [5]: one being constraint satisfaction. Which puts the focus on finding a model(s) which satisfies all constraint. which can be done with generating values within the domain of the variables and testing them, also called generate and test, but obviously this is not the fastest way.

On the other hand we have constraint optimization, which covers an even harder problem. Instead of having to check if a constraints satisfies, here we want to know what model will give us the best value. The optimization function is often called the objective function and occurs quite often in real life problems [5]. Unfortunately depending on the problem we often hit limitations due to NP-hardness. Different search strategies can help to gain a higher efficiency and a lot have been though off. Popular approaches to finding solutions in both branches are the use of: constraint propagation, backtracking, symmetry breaking, dynamic programming, techniques from the SAT solvers like lazy clause generation (more on that later) and even heuristics as: local search, Tabu search, simulated annealing and more.

4.2.2 MiniZinc

A keen observer has will have noticed that we used "after Underlined text a solver has found satisfiability" in the explanation of "send + more = money" 4.2. This is because MiniZinc is not a solver, it came to be from the lack of standard modeling language surrounding CP's. When you wanted to use another solver you had to model you problem again in that solver's specific language. This is what Nicholas Nethercote et al. wanted to solve, they came up with MiniZinc which is a modeling language for CP's that is not connected to a single specific solver [36]. In the words of Peter J. Stucke, a member of the MiniZinc team:

> "MiniZinc is high level enough to express most combinatorial optimization problems easily and in a largely solver- independent way; (...) However, MiniZinc is low level enough that it can be mapped easily onto many solvers." [40]

MiniZinc transforms its inputs to FlatZinc which then can be solved by a growing list of CP solvers. At the time of writing 17 solvers have FlatZinc interfaces. MiniZinc originated from a modeling languages focused on constraints, called Zinc [2]. As you can tell by its name MiniZinc is a subset of Zinc [36].

MiniZinc Challenge

here further

An intentional benefit of having a standardized constraint programming language is the ability to benchmark CP solvers. Which the MiniZinc team did with the organization of the "MiniZinc Challenge"

The challenges is intended as a benchmark but results as well in more solver implementing the FlatZinc API. The challenge even attracted SAT solvers like BumbleBEE [29] which did need an extra translation layer, BEE, to convert FlatZinc to conjunctive normal form (CNF). Although it is limited to finite domain constraint problems.

4.3 SAT

lazy clause generation example?

4.4 SMT

4.5 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 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] Özgür Akgün et al. “Metamorphic testing of constraint solvers”. In: *International conference on principles and practice of constraint programming*. Springer. 2018, pp. 727–736.

[2] Maria Garcia de la Banda et al. “The modelling language Zinc”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2006, pp. 700–705.

[3] Sébastien Bardin, Nikolaj Bjørner, and Cristian Cadar. “Bringing CP, SAT and SMT together: Next challenges in constraint solving (dagstuhl seminar 19062)”. In: *Dagstuhl Reports*. Vol. 9. 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[4] 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.

[5] Roman Barták. “Constraint programming: In pursuit of the holy grail”. In: *Proceedings of the Week of Doctoral Students (WDS99)*. Vol. 4. MatFyzPress Prague. 1999, pp. 555–564.

[6] Dmitry Blotsky et al. “Stringfuzz: A fuzzer for string solvers”. In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 45–51.

[7] Hanno Böck. *How Heartbleed could've been found. Hanno's blog*. English. Apr. 7, 2015. URL: `https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html` (visited on 08/04/2022). 2015-04-07.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Henry Dudeney. "Send+More=Money". In: *Send+More=Money*. Strand Magazine 68 (July 1924), 97 and 214.

[14] 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.

[15] 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.

[16] Eugene C Freuder. "In pursuit of the holy grail". In: *Constraints* 2.1 (1997), pp. 57–61.

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

[18] 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.

[19] Robbert Heatson. *How to write an afl wrapper for any language*. English. July 8, 2019. URL: https://robertheaton.com/2019/07/08/how-to-write-an-afl-wrapper-for-any-language/ (visited on 08/06/2022). 2019-08-07.

[20] 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.

[21] Joxan Jaffar and J-L Lassez. "Constraint logic programming". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1987, pp. 111–119.

[22] Christian Klinger, Maria Christakis, and Valentin Wüstholz. "Differentially testing soundness and precision of program analyzers". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 239–250.

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

[24] Mark H Liffiton and Karem A Sakallah. "Algorithms for computing minimal unsatisfiable subsets of constraints". In: *Journal of Automated Reasoning* 40.1 (2008), pp. 1–33.

[25] 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.

[26] 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.

[27] Kim Marriott, Peter J Stuckey, and Peter J Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

[28] William M McKeeman. "Differential testing for software". In: *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[29] Amit Metodi and Michael Codish. "Compiling finite domain constraints to SAT with BEE". In: *Theory and Practice of Logic Programming* 12.4-5 (2012), pp. 465–483.

[30] 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.

[31] 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.

[32] 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.

[33] Barton P. Miller. *Fall 1988 CS736 Project List*. English. Project List. Computer Sciences Department, University of Wisconsin-Madison, Sept. 1988. URL: `http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf` (visited on 07/28/2022).

[34] 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`.

[35] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[36] Nicholas Nethercote et al. "MiniZinc: Towards a standard CP modelling language". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 529–543.

[37] 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.

[38] Pierluigi Paganini. *Exploiting and verifying shellshock: CVE-2014-6271. The Bash Bug vulnerability (CVE-2014-6271)*. English. Sept. 27, 2014. URL: `https://resources.infosecinstitute.com/topic/bash-bug-cve-2014-6271-critical-vulnerability-scaring-internet/` (visited on 08/04/2022). 2014-09-27.

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

[40] Peter J Stuckey et al. “The minizinc challenge 2008–2013”. In: *AI Magazine* 35.2 (2014), pp. 55–60.

[41] Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. “TestMC: testing model counters using differential and metamorphic testing”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 2020, pp. 709–721.

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

[43] 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.

[44] Patrick Ventuzelo. *Can we find Log4Shell with Java Fuzzing? (CVE-2021-44228 - Log4j RCE)*. English. fuzzinglabs. Dec. 13, 2021. URL: `https://fuzzinglabs.com/log4shell-java-fuzzing-log4j-rce/` (visited on 08/04/2022). 2021-12-13.

[45] Wikipedia. *Constraint programming*. English. Wikipedia. 2022. URL: `https://en.wikipedia.org/wiki/Constraint_programming`.

[46] Dominik Winterer, Chengyu Zhang, and Zhendong Su. “Validating SMT solvers via semantic fusion”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 718–730.

[47] Peisen Yao et al. “Fuzzing smt solvers via two-dimensional input space exploration”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 322–335.

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

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

[50] Chengyu Zhang et al. “Finding and understanding bugs in software model checkers”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 763–773.