KU LEUVEN

FACULTEIT
INGENIEURSWETENSCHAPPEN

Fuzz Testing of Constraint Programming

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. T. Guns

Evaluatoren:
Dr. ir. S. Kolb
Prof. dr. ir. G. Janssens

Begeleider:
Ir. I. Bleukx

Academiejaar 2022 – 2023

Preface

Although this thesis was finished with a strict planning I put onto myself, it was immensely fun to work on. From the destructive nature of fuzzing bugs to the fascinating topic of constraint solving, it all interested me and I had not a single moment where I had to push myself to start working on it. Nevertheless, this thesis would not have been possible without the following people.
Firstly, I would like to thank professor dr. Tias Guns for the guidance and the proposal of this fascinating topic, ir. Ignace Bleukx for answering many questions, intensive thesis meetings, proofreading and the cleverness for coming up with the name of CTORM, dr. ir. Jo Devriendt for finding bugs within our bug finder, the rest of the CPMpy-team, Hakan Kjellerstrand for publishing a significant number of examples which we used as seeds, friends for proofreading even all the way back from industrial engineering on campus De Nayer like ing. Simon Vandevelde. Lastly, I would like to thank my family for the support during my further studies.

Ruben Kindt

Contents

Abstract

This thesis presents a comparative study between three ways of finding bugs in CPMpy as a use case to examine which techniques are suitable for finding bugs in constraint programming languages. With these languages, problems (including logical and optimization) can be modeled and solved. CPMpy extends the modeling part of these problems to be done in a for most familiar programming language python and allows for a close connection to NumPy and the solvers. In CPMpy we will search for bugs where it may crash, hang or produce the wrong output. All of these are unwanted but the last ones do not show the user that a bug has occurred. The first technique to find bugs builds further on an existing paper to test solvers for SMT theories, which this paper converts to be able to test CPMpy with. A second technique uses output preserving equivalent changes to see whether the original result differs from the result of the modified program. The final technique uses the concept of comparing the results of analog programs in order to detect any differentiations between any of the programs.

Of the 23 issues and questions submitted to the repository 19 turned out to be bugs. The first technique found 10 bugs, the second found 13 bugs and the last one found 11 out of the 19 found bugs. Resulting not a single technique able to find all bugs. With more development in constraint programming and CPMpy these techniques would help with finding bugs in the future.

Keywords: constraint programming, CPMpy, fuzz testing, bugs, STORM, metamorphic testing, differential testing

Samenvatting

Deze masterproef stelt een vergelijkende studie voor tussen drie manieren om bugs te vinden met CPMpy als voorbeeld. Dit om te onderzoeken welke technieken geschikt zijn om bugs te vinden in programmeertalen voor beperkingen. De eerste techniek bouwt verder op een bestaande paper om SMT-theorieën te testen via de tester STORM, deze paper converteert STORM naar een tester die om kan gaan met CPMpy om daar dan mee te kunnen testen. Een tweede techniek maakt gebruik van resultaatsgebonden equivalente wijzigingen om te evalueren of het oorspronkelijke resultaat verschilt van het resultaat van het gewijzigde programma. En ten slotte gebruikt de laatste techniek het concept van gelijkaardige programma's, om de resultaten hiervan te vergelijken om zo eventuele verschillen tussen de programma's te detecteren.

Geen van de gebruikte technieken resulteerde in een perfecte score rond de gevonden fouten. De eerste vond 10 fouten, de tweede 13 en de laatste 11 van de in totaal 19 gevonden fouten. Met verdere ontwikkelingen in CPMpy en de programmeertalen zullen nieuwe fouten gemaakt worden en hierbij zouden de technieken gebruikt kunnen worden om deze op te sporen.

List of Figures

List of Tables

List of Listings

List of Abbreviations

Abbreviations

CI/CD	Continuous Integration and Continuous Deployment, a pipeline for newly written code to repeatedly be built, tested, released, deployed and more.
CP	Constraint Programming language sometimes also referred to as CPL
CPL	Constraint 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 variables with a specific domain e.g., boolean, finite and others.
CPMpy	Constraint Programming and Modeling language for Python.
CVC	Cooperating Validity Checker a popular SMT-theorem prover
LLVM	Although it looks like an abbreviation, it is not. LLVM is the name of a project focused on compiler and toolchain technologies.
MIP	Mixed Integer Programming, a theory where decision variables are allowed to be integers
MUS	Minimal Unsatisfiable Subset, the smallest subset possible that is not satisfiable
NNF	Negation Normal Form, which is a boolean formula written using only conjunctions, disjunctions and negations, with negations only occurring in literals.
PUT	Program Under Test, the piece of code, application of program that is tested on for potential bugs.
SMT	Satisfiability Modulo Theory, a generalization of SAT extends to more complex data forms like integers, strings and many more.

List of Symbols

Symbols

$\sim$	Representation for a negation used by CPMpy
&	Representation for a “and” used by CPMpy
|	Representation for a “or” used by CPMpy
$\neg$	Mathematical representation for the logical negation
$\wedge$	Mathematical representation for the logical “and”
$\vee$	Mathematical representation for logical “or”

Chapter 1

Introduction

Finding software mistakes is one of the harder and most time consuming problems a developer faces. There are a lot of causes for bugs: software complexity, multiple people writing different parts, changing objective goals, misaligned assumptions and more. Most of these things cannot be avoided during the creation of software but are the cause of program crashes, vulnerabilities or wrong outcomes. To overcome this, 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 and evaluate previously 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 performed by Miller [45]. Today this technique is seen as random generation based black box fuzzing while the current fuzzing envelops a broader term, as Manès et al. [39] describes,

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

, as quoted from [39]. With this technique we will try to detect bugs in the constraint programming and modeling library CPMpy [29] created by professor dr. Guns et al.

1.1 The usage of fuzzers in the software development cycle

During the development phase of software, tests are performed 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 in multiple different ways, for example by code review, manual testing or automated testing. All these techniques could contain: unit tests, checking for known bugs, regression testing, confirming that the use cases are working, code audits, dynamic testing,

fuzzing and others. None of the techniques mentioned can prevent all possible bugs from occurring and using only a single technique would cost more to find the same level of bugs then using a combination of 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. In other cases dynamic testing may be better, imagine having a program which parses curricula vitae to check if candidates match the job position and you want to check if a fresh computer science graduate fits the position of software analyst. In this case it may be a lot easier to test some curricula vitae 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 program under test (PUT). Here fuzzing may be the correct tool to use. By submitting random inputs into the PUT and looking at the next actions the program takes (i.e., crashes, gives wrong results or others) the fuzzer can automatically detect bugs.

Fuzzing emerged in the academic literature at the start of the nineties, with the industry's full adoption still ongoing thirty years later. Multiple companies like Google, Microsoft and LLVM have created their own fuzzers and together with a pushing security sector for the adoption, have caused fuzzing to become a part of the growing toolchain for software verification.

1.2 Fuzzing and security

Fuzzing is a thirty year old technique to automate the finding of bugs and eventually some of these bugs will be security related. Depending on the application of the program and its environment it is either a problem or not. However, those security related bugs could be costly as discussed in the previous section, bugs become more costly the later you catch them. With security bugs being seen as the pricier ones, i.e., you do not want your company to get hacked, sued or being featured in the news for being exploited on top of the normal cost of having to find and fix the bug. Ideally a company should not have to spend time, energy and money into finding and fixing bugs, but there will be miscommunications, mistakes and more that will result in bugs. Therefore, companies need to invest in prevention and (early) detection. This is shown in the adoption of fuzzers that has gained speed due to its proven effectiveness in finding security related bugs. For example, ShellShock, Heartbleed, Log4 Shell, Foreshadow and KRACK could have been found using fuzz testing as shown in multiple sources [12, 31, 52, 60] and fuzzing is even recommended by the authors to prevent similar exploits [58, 59].

While fuzzing is often used for finding bugs in general, there are even fuzzers that have a focus on catching security vulnerabilities specifically. Fuzzers like the one made by Yuwei Li et al. [37] do this with their Vulnerability-Oriented Evolutionary Fuzzing tool.

1.3 Goals

With this thesis we aim to compare multiple known techniques for automatically finding bugs in constraint programming languages. As this type of language differs from most used programming languages not all techniques may work equally well. On top of that some bugs could have a big impact and it is not always clear that they occurred. The techniques used will be a modified fuzzer originally used on SMT problems, a second will be to apply satisfiable preserving changes to known inputs and test that the original input matches the known input. Our last technique will be taking advantage of the fact that CPMpy has a big library of solvers, which will be used to check that all the solvers agree on the solutions.

1.4 Research questions

As the title of the thesis already may have suggested, we are trying out multiple fuzzing techniques on CPMpy, with the goal of finding which technique works well for this specific type of programming language. This in order to give a push to identify ways of automatically discovering (and maybe solving) new bugs in constraint programming languages. We put forward two regions of research questions this thesis will focus on.

1.4.1 Problem statement

As described in the introduction of this chapter 1, bugs are practically unavoidable and always unwanted, especially when a user trusts a program to give a correct answer and it does not. With solvers surrounding constraint programming languages being executed more and more we would like to strongly avoid any bugs in the real world from arising. To this end, it would be interesting to find bugs during development without much overhead, a modern approach would be the use of fuzzers, which we will try out on a constraint programming language.

1.4.2 Main focus: fuzzing technique-focused

The first and our main focus will be comparing different fuzzing techniques. We are going to modify a successful SMT fuzzer STORM to the CPMpy language, which we will name CTORM for CPMpy STORM. Secondly we will use the technique of metamorphic testing and lastly try finding bugs with differential testing. Resulting in the following research questions (RQ):

1. RQ1: What fuzzing technique will find the most bugs?
2. RQ2: What fuzzing technique will find the most critical bugs?
3. RQ3: What type of bugs will be found by each fuzzing technique?

1.4.3 Classification-focused

Our next and last focus will be on the classification of found bugs, giving us the following research questions.

1. RQ4: How many (critical) bugs can we find?
2. RQ5: What are the causes of the bugs?

1.5 Modus operandi

This thesis will create or adapt three techniques to detect bugs automatically for the constraint programming language, CPMpy. Starting from multiple examples we will extract all solvable models to then run each model through a technique to see if we can create a bug either from the original model or by modifying it. Afterwards, we will validate and evaluate the found bugs for each technique. To finally compare the results and see which techniques are able to find bugs in constraint programming languages.

1.6 Overview

In the next chapter we will go deeper in on CP, SAT and SMT-theories to then explain what fuzzing is in Chapter 3 and how we can use it to find bugs. Once we found bugs we will need to find the relevant or crucial part related to the bug, which we will discuss in Chapter 4. We will also explain how we implemented the techniques to find the bugs in more detail in Chapter 5 and which ones we found in Chapter 6. To then end with a conclusion and future work in Chapter 7.

Chapter 2

CP, SAT and SMT

2.1 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."
> -Eugene C. Freuder in "In Pursuit of the Holy Grail" [24].

As the quote of Eugene C. Freuder says, he and others believe that in the ideal world the user conveys any problem to a computer or a program and that the program will solve it, which matches a coarse summary of what a constraint programming language (CP) can do in an ideal world. Unfortunately, we do not live in an ideal world, problems need to be converted or split up into a representation that the solver can understand. Here constraints come into the picture, constraints are mathematical, logical or relational connections put on or between variables to form a model that hopefully satisfies all constraints after solving. This is sometimes combined with a final objective constraint to be minimized or maximized, for example finding a model where a postman visits all cities on a route but with a minimal distance traveled. With CP the focus lays more on solving high level problems with specialties around scheduling and planning [9], CP's key feature being the global constraints, a set of "functions" that are aimed at a high level of solving. For example, the `alldifferent()` which makes sure that all variables within an array will be assigned a different value or `circuit()` that holds if the input forms a Hamiltonian circuit.

A second field of research that we will discuss is that of 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 efficient solving [5].

Finally, we will look at 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 [5, 49].

2.2 Constraint programming

As mentioned before CP's are versatile in solving constraints, especially when it comes to planning and scheduling. This by their efficient constraint propagation, backtracking and the linking of related constraints [62]. Originally CP's can be linked back to constraint logic programming (CLP) where programming languages (mostly logic programming languages) were combined with constraints and ways to solve them. A notable version is that of Joxan Jaffart and Jean-Louis Lassez [32, 62] with their extension in Prolog and thereby the creation of the category.

To further introduce constraint programming we will show a popular puzzle in the CP field "send more money" as seen below. It is a logic puzzle made by Henry Dudeney and published in Strand Magazine's July 1924 edition [21]. In this puzzle each character represents a single digit between zero and nine (both included), meaning that the 'e' from "send" should be the same digit as the 'e' from "more" and the 'e' from "money". The other rules are as follows, all different characters should have a different digit, any of the starting letters of each word cannot be zero and after replacing all characters with their corresponding digits the sum should be correct.

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

The "send more money" puzzle.

On top of showing the problem we will also use MiniZinc[1] as our constraint programming language to find the solution for this puzzle. By doing this we will show a possible representation of these types of problems using constraints so that a solver can find a solution.

On the first line of Listing 2.1 you can see the importing of the global constraints, which CPs are known for. On line 2 until 9 we can see the declarations of all possible characters to the possible digits: starting letters such as '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 the semicolon. This constraint specifies the matching of the sum. At line 16 we use one of the imported global constraints namely `alldifferent()` which will satisfy if no duplicate values occur in the array. To then start solving for satisfiability at line 18, this in comparison of solving for minimization or maximization an objectify function, which does not apply here. Finally, after a solver has found a solution we print the result using a

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

pretty-print from line 19 to 22. We will not be spoiling the solution and will leave it up to you to find the solution but remember that you can check your answer with the program below.

```
1  include "globals.mzn";
2  var 1..9: S; %'S' is a starting character therefore limited from 1 to 9
3  var 0..9: E; % Other characters are limited from 0 to 9.
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;
19 output % Pretty-printing the solution.
20 [" \(S)\(E)\(N)\(D)\n",
21 "+ \(M)\(O)\(R)\(E)\n",
22 "= \(M)\(O)\(N)\(E)\(Y)\n"];
```

Listing 2.1 – Solution to the puzzle "send more money" modified and taken from `https://www.minizinc.org/doc-2.5.5/en/downloads/send-more-money.mzn`.

The "alldifferent()" and the sum constraint also show us the power of CPs, as with a single statement multiple relations are expressed [41]. Instead of needing to specify all possible relations of the last three characters of each word (D + E = Y, Y - D = E and Y - E = D) a single expression suffices. When knowing two values we can infer the third. It saves a significant amount of work for the developer, definitely when it comes to the bigger `alldifferent()` constraint. This in combination with the efficient back-end solvers for this constraint allows for a bigger abstraction and makes MiniZinc and other CP-solvers so powerful.

2.2.1 Origin

Within constraint programming we can distinguish two branches [9], one being constraint satisfaction, which puts the focus on finding a model which satisfies all constraints. This can be solved with generating values within the domain of the variables and testing them, also called generate and test but this is not the fastest way. The second being constrained optimization, which covers an even harder problem. Instead of having to check if all constraints satisfy, we want to know what model will give us the highest or lowest value. The function to optimize is often called the objective function and it occurs often in real life problems [9], since:

> "(...) we do not want to find any solution but a good solution."
> -Roman Barták from "Constraint Programming: In Pursuit of the Holy Grail" [9]

Unfortunately depending on the problem, we regularly hit limitations due to NP-hardness. This has challenged the field and multiple different search strategies to gain a higher efficiency have been thought off. Popular approaches to finding solutions in both branches are the use of constraint propagation, backtracking, symmetry breaking, dynamic programming, techniques from the CP solvers like lazy clause generation and even heuristics such as local search, Tabu search, simulated annealing and more.

2.2.2 MiniZinc

A keen observer will have noticed that we used "after a solver has found a solution" in the explanation of "send more money" in Section 2.2. This is because MiniZinc is not a solver but a modeling language, it came to be from the lack of standard modeling languages surrounding CP's. Before MiniZinc, when you wanted to use another solver, you had to rewrite your problem again in that solver's specific language. This is what Nicholas Nethercote et al. [50] wanted to solve when they came up with MiniZinc. A modeling language for CPs that is not connected to a single specific solver. It originated from a modeling language focused on constraints, called Zinc [3] of which MiniZinc is a subset of [50]. 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."
> -Peter J. Stuckey et al. in "The MiniZinc Challenge 2008-2013" [56].

which shows the team's vision of MiniZinc.

MiniZinc transforms its inputs combined with the model, data and solver specific features to FlatZinc, an intermediate language created for easier interpretation by solvers. Which then can be solved by a specific solver.

MiniZinc Challenge

On top of maintaining and improving MiniZinc, its team also organizes a yearly challenge to compare and test what improvements have been made in the constraint solving world, which they are able to do by having the benefit of a standardized constraint programming language to benchmark with. In this yearly challenge each solver gets fifteen minutes to solve a hundred selected problem instances. However, due the difficulty of having to find quite a number of good representative problem instances each year, the organizers of the challenge ask the participants to submit preferably two problem models and multiple related instances. From the received list the jury then tries to make a fair selection to cover the use of global constraints, real-world representative problems, to find a good balance between satisfying versus optimization problems and the different types of technologies (SAT and MIP) instead of only selecting CP focused problems [56].

It is due to this MiniZinc challenge that a better connection has formed between CP solvers and SAT solvers. By attracting SAT solvers to the challenge, tools such as fzn2smt [13] and BEE [43] arose. These tools are able to translate FlatZinc to SAT-LIB and conjunctive normal form (CNF) respectively. With BEE producing CNF, Amit Metodi and Michael Codish were then able to let a SAT solver solve the problem [43]. Although the latter is limited to finite domain constraint problems, it allows utilization of large and swift SAT-solvers on top of bringing the field of research closer to each other.

On top of being a great way to benchmark comparative solvers, the challenge also results in more solvers that implement the FlatZinc as an input and, due to the competitive nature of academics, brings the motivation to stride forwards [55].

2.2.3 CPMpy

MiniZinc is not the only tool with the idea to create a modeling language for CP's. Other examples are Essence with the focus on combinatorial problems for people with a background in discrete mathematics [25]. Additionally, the one which piques our interest is the constraint programming and modeling language for Python (CPMpy). Based around the popular packet NumPy, CPMpy allows for a lower learning curve of constraint programming languages by creating a constraint programming and modeling language in a familiar language, Python, [29]. NumPy comes with a lot of advantages, it is popular in data processing and the general array-based operations among machine learning and more. Being able to use these in combination with CPs would allow for both fields to grow closer to each other.

At the time of writing CPMpy has support for multiple solvers like OR-tools, CP-SAT, Gurobi (for MIP problems), PySAT (which is a library that contains 13 different SAT solvers), PySSD (which is a knowledge compiler). On top of these, CPMpy has support for any CP-solver that support text-based MiniZinc language [27, 28], resulting in at least 33 extra solvers. With potential future extensions to Microsoft's Z3 theorem prover and others as new solvers. We refrain from discussing CPMpy in-depth explanation as it is in active development at the time of writing and would otherwise become quickly outdated. We do look forward to seeing CPMpy progress and the future paper(s) discussing it. What we can do is discuss CPMpy on a global level, where we see CPMpy existing out of four components. This being a model part (everything to do with creating a model of a problem), a transformation part (everything to do with changing constraints), a solver interface (everything to do with restrictions the solver puts on CPMpy) and finally the solvers themselves (these solves the final model), with CPMpy having little to no control over the last one as can be seen in Figure 2.1.

Finally, to illustrate that both MiniZinc and CPMpy are not that far apart a CPMpy Listing 2.2 is included for the same problem as seen for MiniZinc in Listing 2.1. After importing the relevant packages on line 9 and 10, the variables are created with a lower bound of 0, an upper bound of 9 and since there are 8 variables an array of length 8 is generated. Next on line 14 an empty model is generated which is filled with constraints on lines 15 to 20. All letters are calculated with their corresponding

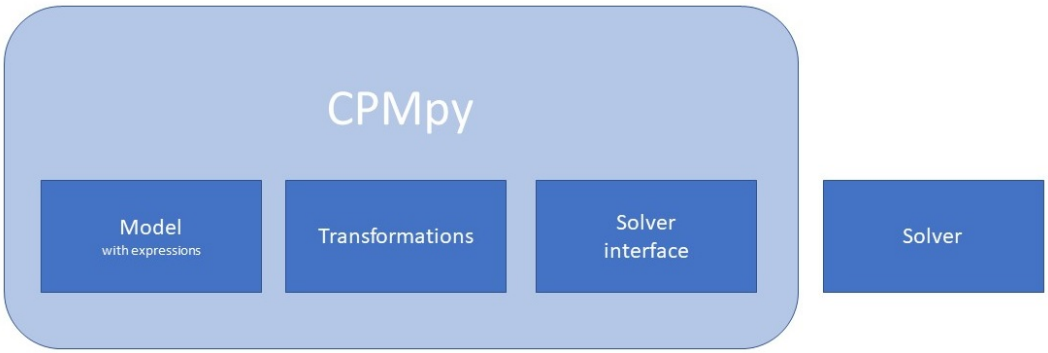

Figure 2.1 – The four main components of CPMpy

units to for the summation constraint, the next line specifies that all unique letters must get a unique value assigned. To then limit the words starting with a zero by constraints on line 19 and 20. After a print of the model to show what has been modeled the model is solved on line 25 to then pretty print the output.

```
1  #!/usr/bin/python3
2  """
3  Send more money in CPMpy
4    SEND
5  + MORE
6  ------
7   MONEY
8  """
9  from cpmpy import *
10 import numpy as np
11
12 s,e,n,d,m,o,r,y = intvar(lb=0, ub=9, shape=8) # creating variables
13
14 model = Model() # with "+=" we can add a constraint to the model
15 model += sum([s,e,n,d] * np.array([1000, 100, 10, 1])) \
16        + sum([m,o,r,e] * np.array([1000, 100, 10, 1])) \
17       == sum([m,o,n,e,y] * np.array([10000, 1000, 100, 10, 1]))
18 model += AllDifferent([s,e,n,d,m,o,r,y])
19 model += s > 0 # in MiniZinc each variable was declared separately in CPMpy
20 model += m > 0 # we can do it in batch, resulting in these extra constraints
21
22 print(model)
23
24 # Solve and pretty-print
25 if model.solve():
26    print("  S,E,N,D =   ", [x.value() for x in [s,e,n,d]])
27    print("  M,O,R,E =   ", [x.value() for x in [m,o,r,e]])
28    print("M,O,N,E,Y =",    [x.value() for x in [m,o,n,e,y]])
29 else:
30    print("No solution found")
```

Listing 2.2 – Solution to the puzzle "send more money". Modified from the example in the CPMpy repository `https://github.com/CPMpy/cpmpy/blob/master/examples/send_more_money.py`

2.3 SAT

Now that we discussed CPs, let us step back to boolean satisfiability problems (SAT). Here we are focused on checking the satisfiability of boolean formulas. SAT has been successful in the hardware design and verification. With a lot of research in improving SAT-solvers, they have become significantly efficient on large problems [5]. These efficiency improvements come from the DPLL algorithm to conflict-driven clause learning (CDCL) combined with the addition of non-chronological back jumps. Whereas, better propagation, lazy clause generation (LCG), data structures and the introduction of heuristics assisted significantly to the process. Heuristics often include clause deletion to decrease the number of unused clauses, variable state independent decaying sum (VSIDS) where we add a decaying weight (the sum) to select which literal we prioritize, random restarts where we try to avoid large search trees by restarting with the learned clauses, among other heuristics [18, 33, 54].

2.4 SMT

An extension of SAT is satisfiability modulo theories (SMT), which extends the only boolean focus of SAT with quantifiers, integers, lists, arrays and much more. With the advantage that most efficient algorithms from SAT transfers over. The focus with this technology lies in multiple fields but it is quite popular in program verification and testing. This can be seen with Microsoft's Z3[2], which is focused on static software checking [49]. Z3 is one of the most popular SMT theorem provers at the moment, with a considerable number of varying supported theories. It has support for the standard SMT-LIB input, the simplify input (from an older theorem prover) [19], and a low-level native input for textual input-based directly. Z3 also has APIs for the programming languages like C/C++, .NET, OCaml, Python, Java, Haskell and more [63].

The second most popular SMT-theorem prover is from the cooperating validity checkers (CVC) line-up. The latest in the CVC-lineup is CVC5[3], which is the fifth in line of the CVC-family, the previous being SVC (not always counted), CVC, CVC Lite[4], CVC3[5][7] and CVC4[6]. Like CVC4, CVC5 supports the standard SMT-LIB input format among other formats directly and has APIs for C++, Python, Java and potential others in the future [4, 8].

2.5 Conclusion

In this chapter we discussed some of the most popular parts of constraint solving. We explain the boolean constraints within SAT to be able to better explain the

[2] `https://github.com/Z3Prover/z3`
[3] `https://github.com/cvc5/cvc5`
[4] `https://cs.nyu.edu/acsys/cvcl/`
[5] `https://cs.nyu.edu/acsys/cvc3/`
[6] `https://cvc4.github.io/`

broader non-boolean constraints within SMT and constraint programming. On top of that have also seen multiple tools used in the industry and academia to prove and/or solve constraint problems.

Chapter 3

Fuzzing

The rise of fuzzing came with Miller giving his classroom assignment [47] 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 [45] 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 prevalent classifications made, what the fuzzer expects as input, what we can expect as output and we will look at the most popular fuzzers.

3.1 Classifications

The three most popular classifications are: how does the fuzzer create input, how well is the input structured and does the fuzzer have knowledge of the program under test (PUT) [26, 36, 39].

3.1.1 Generation and mutation

A fuzzer can construct inputs for a PUT in two ways, it can generate input itself or it can modify parts from an existing input, called seeds. 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 caused by the fact that generating semi-valid input becomes a lot harder the longer the input is. For example, generating the word "Fuzzing" by uniformly random sample of ASCII symbols, 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 then make modifications to create semi-valid inputs. This last technique the diversity of the seeding inputs does become quite important. Ideally, we would have an unlimited diverse set of inputs, but that may not always be available. A workaround to this issue is proposed in the paper by Alexandre Rebert et al. [53]. 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.

3.1.2 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 input structure. This technique can be applied similarly to mutational fuzzing as well. The main difference to generational fuzzing is that in mutational fuzzing we can also remove or or change randomly selected symbols.

There are three types of inputs: non-valid, semi-valid and valid inputs. With non-valid inputs we will almost be exclusively testing the lexical and syntactic stage of the PUT, which often comes down to just the parser. Either the input crashes the parser or it will be detected 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. Lastly, with valid input we are testing if the PUT behaves as expected and does not crash, although we cannot know which type a given input is before giving it to the PUT.

A smart fuzzer refers to the fuzzing 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). The latter one can for example match all parentheses, while the former would be able to create correct keywords such as "continue" for example. Directed fuzz testing, where we guide the fuzzer on a specific code location via an explicit 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.

3.1.3 Black, gray and white box fuzzing

On top of adding knowledge of the inputs' structure to the fuzzer, we can also add knowledge of the program under tests' structure to the fuzzer, which brings us to black, gray and white box fuzzing. Black box fuzzing does not have any knowledge about the inner workings of the PUT compared to white box fuzzing. We provide input and we look at what the PUT provides as output, 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 white box testing is the term used when the fuzzer has as much information available as possible. It will have access to the source code and can adjust their

inputs to fuzz specific parts of the code. Directed fuzzing also falls under this term. Here we guide the fuzzer to interesting locations for testing of specific parts of the PUT. White box fuzzing does have a higher computation cost due to having to reverse engineer the path to specific edge cases, meaning that it has a higher chance of finding more bugs per input but creating those inputs takes more time compared to black box fuzzing.

The differences between black, gray and white box fuzzing are 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 could become controversial, all we can say is that it is not a black-and-white situation and that the lines have become fuzzy.

3.2 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. It 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 and/or non-printable ASCII, with or without null-terminating characters of a random length. That resulted in a successful paper two years later [45].

His later work in 1995 on even more UNIX utilities, X-Windows servers [46] in 1995, Windows NT 4.0 and Windows 2000 [23] in 2000, MacOS [48] in 2006 and his recent revisit in 2020 on fuzzing [44] all fall in the same category of random generation based black box fuzzing. His papers showed that a significant portion of programs can 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 [16] was developed by Cadar et al. KLEE is a generation based white box fuzzing tool built with the idea that bugs could be on any code line and that testing should cover as much code as possible. A code coverage tool is used to test which lines of code are executed and this combined with the feedback KLEE received from the symbolic processor and the interpreter it can generate improved inputs. KLEE does this by symbolically executing the program executions, branching on all paths and searching for any dangerous operations. When it finds an error, it will convert the symbolic representation to a concrete representation based on the constraints it needed to get to the specific location. To then use this concrete representation to test the original program.

With KLEE's stride to obtain a high 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 is sometimes used as a relative metric, checking if a specific

test raises the code coverage. This 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 get to 100% turns code coverage into a well-rounded measurement.

Figure 3.1 – On the left a young American Fuzzy Lop Rabbit used as the name for AFL and AFL++ and on the right AFL++'s logo. Images taken from animalcorner.org and AFL++'s website respectively [2, 61].

As for the more popular fuzzers, American fuzzy lop[1] (AFL), which is named after an American rabbit breed (see Figure 3.1) and is a C and C++ focused mutation based gray box fuzzer released by Google. However, due to inactivity on Google's part the fork AFL++[2] has become more popular than the original and is maintained actively by the community [22]. Not only is it actively maintained, but it is also actively used by researchers and the industry. Besides sparking the existence of AFL++, AFL has also triggered a python[3] focused version, a Ruby[4] focused one and a Go[5] focused version. Robert Heaton [30] showed that it is not difficult to write a wrapper for it.

A potential reason for the inactivity of Google on the AFL project could be the development of both Clusterfuzz[6] and OSS-fuzz[7], respectively a scalable fuzzing infrastructure and a combination of multiple fuzzers. The former one being used in OSS-fuzz as a back end to create a distributed execution environment, which had quite a bit of success according to both the CusterFuzz and the OSS-Fuzz repository respectively [20]:

[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/`

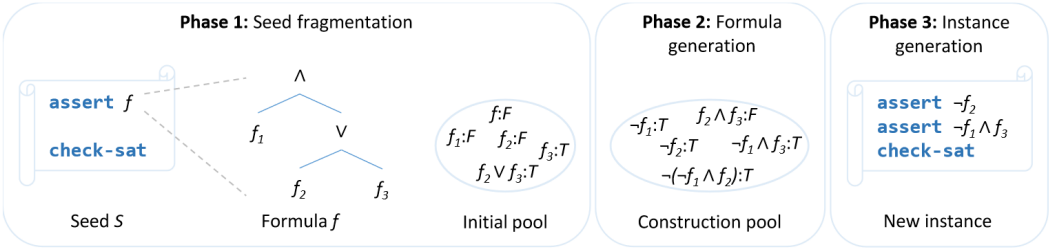

Figure 3.2 – Overview of the three STORM phases as presented by Muhammad Numair Mansur et al. in "Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing" [40].

> "As of May 2022, ClusterFuzz has found 29,000 bugs in Google (e.g. Chrome) and 36,000+ bugs in over 550 open source projects" and "As of July 2022, OSS-Fuzz has found over 40,500 bugs in 650 open source projects."

Google is not the only one to come forward with a fuzzer. Even Microsoft has jumped on the bandwagon of fuzzing with its OneFuzz[8], a self-hosted Fuzzing As-A-Service platform which is intended to be integrated with the CI/CD pipeline. However, looking at the given stars on the GitHub repository, it looks like Google's tools are more popular than Microsoft's ones. The last prominent fuzzer we will 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.

3.2.1 Testing CP and SMT with Fuzzers

Until now we discussed fuzzers more generally but now, we would like to focus on specific fuzzers built 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. [40] to find critical bugs (i.e., wrongly sat or wrongly unsat) in SMT solvers. In their paper [40] they explain the inner workings thoroughly, but briefly summarized STORM creates an initial pool of smaller formulas from existing formulas found in seeds and uses another solver to create models of those smaller formulas. To then construct more complex formulas with knowledge of their ground truth, with this STORM can test the SMT solver as represented in Figure 3.2. This novel way of fuzzing SMT solvers with inputs that are satisfiable by construction and has been cited significantly, considering that it is a recent paper.

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

Another technique for fuzzing SMT solvers is the one proposed by Dominik Winterer et al. with their fuzzer YinYang [64], 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."
> -Dominik Winterer et al. in "Validating SMT Solvers via Semantic Fusion" [64].

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 $\phi_1 \equiv \mathrm{X} > 10$ and $\phi_2 \equiv \mathrm{Y} < 9$ with the fusion function for $\mathrm{Z} = \mathrm{X} + \mathrm{Y}$ would become $\phi_3 \equiv (\mathrm{Z} - \mathrm{Y}) > 10 \wedge (\mathrm{Z} - \mathrm{X}) < 9$, 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 [64]. The results of the paper were 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 such as 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 will 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. [65] found quite the success by being the first to link the configuration options to the operations and to then use this information to fuzz better. When using STORM as the underlying fuzzer with the knowledge of the configuration space the authors managed to increase the code coverage by up to 18.8%. Knowing that SMT solvers such as Z3 and CVC4 contain more than 700,000 and 100,000 lines of code respectively means that any percentage is a significant number of extra lines covered.

3.3 Types of bugs

Not only can we classify fuzzers, but we can also classify the types of bugs found by the fuzzers, as done in a recent paper [40] by Muhammad Numair Mansur et al. being crashes, wrongly satisfied, wrongly unsatisfied or hanging inputs. With some of these bugs being less acceptable than others. For example, as Muhammad Numair Mansur et al. describe, 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 in Subsection 3.4.1). Meaning that the user will treat the result (wrongly)

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

as correct, comparing this to a crash where it is clear that something went wrong is more transparent for the user.

With hanging inputs, the user cannot draw incorrect conclusions and with wrongly satisfied models, the user could check the model's instances and confirm the result before using it further, this is due to problems frequently being NP-hard, meaning they are easy to confirm but hard to solve. To avoid the halting problem, we will be treating models that take more than 15 minutes as a hanging model. We can do this since we know that the models are solvable within a minute, but more on that in Section 5.2.

Furthermore, we will treat the wrong amount of solutions as one of the critical bugs, together with the wrongly satisfiable and wrongly unsatisfiable solution. 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" [51] would classify, is the bug in the syntactical or in the semantical part of the program?

3.4 Other forms of testing

3.4.1 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. The latter one uses a single or multiple analog programs to test if the PUT gives the same output as analog programs. By applying "bugs-as-deviant-behavior" Christian Klinger et al. tried to find precision and soundness mistakes in their paper [35], which they did with input creation via seed files to compare the output to similar programs. Unfortunately, neither crash based nor differential testing is ideal: crash based fuzzing will not trigger on wrong outputs and differential testing requires that one or multiple analog programs exits preferably with a different implementation to reduce overlapping bugs. Therefore, The latter technique may not always be possible due to the existence of those analog programs.

3.4.2 Metamorphic testing

In situations where the existence of analog programs would be a limited factor, metamorphic testing could be a solution. A nicely worded definition would be:

> "Metamorphic testing (...) involves generating new test cases from existing ones, where the expected result of a new test case can be generated from the result of an existing test via a metamorphic relation. By comparing the results of the original test with the new one we can identify cases

> where the metamorphic relations are broken (...)".
> -Özgür Akgün et al. in "Metamorphic Testing of Constraint Solvers" [1].

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" [57] the authors add an extra restriction to a variable in a formula to test if the number of models reduces (or remains equal). Furthermore, the authors also tested whether the number of models remained the same compared to the original when given an equivalent formula. This technique no longer depends on a secondary oracle but it does depend on multiple executions and could miss a bug that occurs in multiple situations. In addition, it has been applied in combination with differential testing to test constraint solvers with success by Akgün, Özgür et al. [1], in which they express how well this technique fits with testing constraint solving, due to the availability of metamorphic transformations.

3.5 The oracle problem

The oracle problem describes the issue of knowing if a PUTs output was, given the input, correct or not, as expressed in "The Oracle Problem in Software Testing: A Survey":

> "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."
> -Earl T. Barr et al. in "The Oracle Problem in Software Testing: A Survey" [6].

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 contain all the possible encoding of specifications like modeling languages UML, Event-B and more. Their derived test oracle 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 knowing that crashes are always unwanted and the human oracle such as crowdsourcing respectively.

3.5.1 Handling the oracle problem

Although the approach by Bugariu and Müller in "Automatically testing string solvers" [15] falls in the first category because it concerns a black box fuzzer, their approach is innovative. Most fuzzers either use crashes or differential testing to find bugs, even though they know the (un)satisfiability of their formulas by the way 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 an equivalent but different formula for A, to create the trivial unsatisfiable formulas.

To increase the complexity of those trivial formulas, they again depend on satisfiability preserving transformation, which has also been applied to SMT solvers by Muhammad Numair Mansur et al. called STORM [40]. This 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 means that STORM does not need an oracle to test the entire theory, but only the smaller sub-formulas.

3.6 Opinions against Fuzzing

We have talked about the successes of fuzzing, but there are also voices against fuzzing, as William M. McKeeman [42] 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)? Furthermore, fuzzing seems to generate an infinite number of bugs in the eyes of developers, which is also a pet peeve according to McKeeman. Although we have a bias due to writing this thesis about fuzzing, we believe that those perspectives should be acknowledged in this paper. However, this is not a single view shared by all developers from the papers [64, 65, 68] and in others we see a positive response to newly found bugs by the respective developers. For example, some have even started implementing fuzzers [11] in their toolchain to detect bugs.

3.7 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.

Chapter 4

Detecting Crucial Parts of Inputs

When detecting that the program under test crashes, hangs, wrongly satisfied, wrongly unsatisfied or gives the wrong number of solutions on a given input, the first question to ask is why it would do that. What causes this unwanted output and on what line does the bug occur?. With crashes, a stack trace and some luck this could be easy, but when a bug causes a crash in another place the developer may need to debug deep into the code to find the bug. Finding the bug in potential large inputs could be a tedious and long assignment, for this reason we would like to know what parts of the input are relevant for the bug, which will be discussed further in this chapter, starting with deobfuscating inputs.

4.1 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."
> -Richard Stallman and Ralf Hildebrandt in "Simplifying and Isolating Failure-Inducing Input" [67].

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 on the other hand, can take a lot of trial and error to see which variations still reveal the bug or having to walk through the execution to find the bug. Both take time and if we want to go to absolute minimal inputs, but for developers it is not needed to go to that extreme. As long as the bulk of the unrelated parts of inputs are gone 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 in Subsection 4.3.1) which is also useful information for developers. To find crucial parts of inputs, it is often achieved either with simplification or isolation.

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 4.1 – A minimizing delta-debugging algorithm as shown in “Simplifying and isolating failure-inducing input” by Andreas Zeller and Ralf Hildebrandt [67].

4.1.1 Simplifying

Simplification is the technique where we repeatedly remove parts of a failing input and check if it still fails, which is often done via the “delta-debugging” algorithm, which belongs to the divide-and-conquer family of algorithms [14]. The algorithm can be seen in Figure 4.1 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. Lastly, “1-minimal” means that not a single character can change without the input going from failing to passing. We start the algorithm with the input and split it into two. If we can find a subset that still fails on its own, then we continue with that subset else we look for a subset where the complement of the input still fails but where a subset is missing from the input. In the case where we split the input in two parts this would be the same as the previous result. In case we do not find any smaller subset to continue, then we reduce the granularity of the split by a factor of two. To finally end when it is no longer possible to remove any part of the input. We then have obtained an input where all parts are necessary to expose the bug. The input is then 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 4.2. In the first two steps no removal of any part nor complement was possible therefore we reduced the granularity ‘n’, 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, repeat our possible steps and reach a minimal input.

4.1.2 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 it. Fortunately, this comes with the advantage

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 4.2 – A minimizing delta-debugging example as shown in "Simplifying and isolating failure-inducing input" by Andreas Zeller and Ralf Hildebrandt [67] with an input that is deobfuscated with the ddmin() algorithm from Figure 4.1.

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 by tracking multiple inputs and bigger inputs often take longer to process. However, according to Andreas Zeller et al. [67] this technique is the fastest one of the two techniques. Figure 4.3 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 between the last passed and last failed tests. Additionally, the asterisk '*' indicates that the result is already known and does not need to be recalculated.

4.1.3 Connection with minimal unsatisfiable subset and maximally satisfiable subset

For readers that are familiar with the SMT or constraint solving-world will have noticed that these techniques feels similar to the way of finding the minimal unsatisfiable subset (MUS), which it is, in the case of a solver wrongly stating that an input is

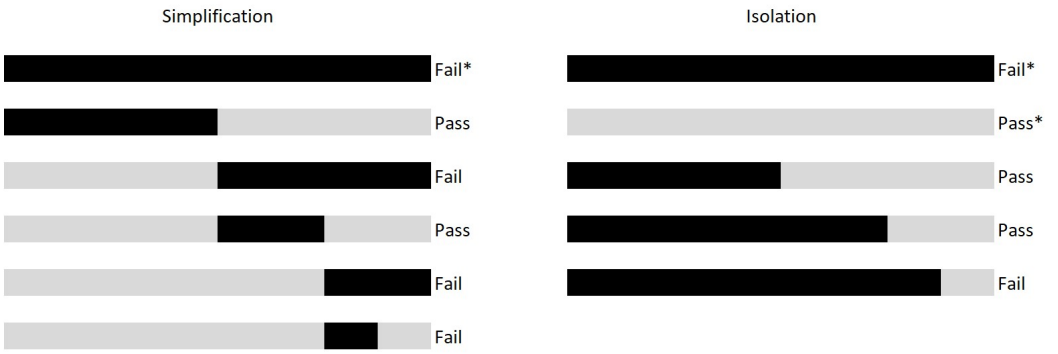

Figure 4.3 – Deobfuscating inputs based on simplification (left) and isolation (right) on the same input. Figure based on an illustration found in "Why programs fail: a guide to systematic debugging" by Andreas Zeller [66].

unsatisfiable. With MUS we try to find the smallest subset of formulas or constraints that will result in an unsatisfiable solution while with maximally satisfiable subset (MSS) we would be trying to find the biggest subset of formulas or constraints that would result in a satisfiable solution. Both are an iterative process and can be applied in the simplification and/or the isolation process. However, searching combinations of formulas results in a smallest subset is a computationally intensive process. Fortunately, a lot of thought has already been put in it to improve it, for example Mark H. Liffiton et al. have proposed multiple "Algorithms for Computing minimal unsatisfiable subsets of Constraints" [38]. In which they discuss their novel sound and complete algorithms for finding all minimal unsatisfiable subsets. Again, we should note that a smallest subset is not needed as we aim to reduce the input to help the developer find the error faster, a difference between a smaller and the absolute smallest subset is a big computational difference in practice.

4.1.4 Alternative approach to deobfuscating

An alternative approach compared to the already mentioned techniques is one proposed by Alexandra Bugariu and Peter Müller [15]. In which they forgo the need of deobfuscating inputs by generating inputs "small by construction". Because the smaller the inputs are, the less space there is for remaining formulas or constraints to obfuscate the input. On the other hand, the chance of finding bigger bugs with multiple constraints interacting with each other will become smaller. A last alternative approach would be retrying fuzzing the same seed after that specific seed has produced an input resulting in a bug, this by adding an increasing size limitation in order to find the same bug with a smaller and smaller input, as done by Muhammad Numair Mansur et al. in "Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing" [40].

4.2 What size to change

A subject we glossed over so far is the chuck size, the size to remove while trying to find the critical parts of inputs. The previously seen techniques will work well on the original fuzz testing by Miller et al. [45] since those random generated symbols were independent from each other. However, when testing more complex words such as function names, we no longer can split on all possible places, since the input would most likely no longer parse.

In Figure 4.3 we conveniently took one-eighths 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. 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 [66]. On top of that, it will make sure that we eliminate all unrelated parts with the specific granularity of two and get "ooXoooo" instead of "ooXooXXoo".

For more complex inputs we can apply techniques seen in Section 3.1.2 where we discussed the creation of randomly and smarter created inputs. Instead of removing 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 [66]. Both lexical (the meaning of words) and syntactic knowledge (the meaning of word combinations) can be used to help us in deobfuscating complex inputs, where syntactic knowledge would help us remove the most since it is the bigger of the two.

4.2.1 Preserving satisfiability

With the techniques mentioned in Section 3.5.1, "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 unsatisfiable and the PUT says it is a satisfiable problem, then we cannot remove constraints to retest, as potential changes could switch the original input from an unsatisfiable to a satisfiable problem. We could use a trusted solver to make sure that we do not change the ground truth by retesting each change as Brummayer and Biere [14] did or as done by Muhammad Numair Mansur et al. [40] try to fuzz the same seed in the hope to find a smaller input that gives the same bug.

The other scenario, where the ground truth says that an input should be satisfiable 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 such as simplifying, isolation, MUS, MSS and the technique of re-fuzzing such as STORM did, while still preserving the (un)satisfiability of the problem.

4.3 Unexpected advantages of deobfuscation

4.3.1 Deduplication of bugs

With deobfuscating the inputs we can detect exact copies, but depending on the deobfuscation's time complexity other techniques could do better with similar results. In cases where we would have access to stack traces, we could differentiate the bugs on the basis of the hash from the backtrace or even numerous hashes per input depending on the number of backtrace lines taken to hash. "Stack backtrace hashing" as this technique is called, is quite popular according to Valentin J.M. Manès et al. [39].

Another technique talked about in "The Art, Science, and Engineering of Fuzzing: A Survey" [39], 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. Microsoft uses this technique [17], which is called "semantics based deduplication", where instead of backtraces they use memory dumps to hopefully find the origins of bugs. However, this use of dumps is less ideal due to traces having more information, but the latter is not always possible due to traces often being removed in production for performance and privacy reasons as specified in the paper.

A last technique for deduplicating bugs would be looking at the bug description left by manual user bug reports, although this depends 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 due to the limited information given in the output.

4.3.2 The precision effect

Finding bugs needs to be done in a careful way, so that we do not change a null pointer dereference bug to a parser related bug. Since, as discussed in the previous chapter, we value some bugs higher than others. In a paper by Andreas Zeller and Ralf Hildebrandt [67] 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. Nevertheless, we try to avoid this effect, which can be done with the techniques in the previous section. However, compared to the previous section where we try to match bugs, here we try to detect if we get the same bug as the last time.

4.4 Conclusion

In this chapter we discussed why a deobfuscated input would be convenient for the further process, what advantages it brings while fuzzing PUTs. We have seen

multiple methods of deobfuscating inputs from the straightforward simplifying to the heavier isolation. We also looked at state-of-the-art approaches, how they preserve satisfiability and how to avoid having to deobfuscate inputs in the first place. To then end with advantages of deobfuscated inputs.

Chapter 5

Implementation

In this chapter we will discuss how we build our fuzzers, what issues we had to circumvent and how we did that. Starting off with how we got our seeds to fuzz upon, to then discuss how we implemented the three techniques to finally end with how we deobfuscated the found bugs.

5.1 Software versions used

Throughout this paper we used CPMpy[1] version V0.9.9 (commit e79b3af), unless specified otherwise, this version was chosen as it was the latest release version at the time of testing the first technique. All techniques were developed in Python 3.8, the MiniZinc solvers came with MiniZinc Python[2] release version 0.7.0 (commit a195cf6). For the proprietary solver Gurobi[3], we used its Python version 9.5.2 with an academic license. Originally, we did try to utilize the trial version to ease possible reproducibility, but the restrictions on the complexity of the problems became a hindrance which resulted in us moving to an academic license. For the other versions of the solvers, we used the ones included in the already mentioned packages, except for MiniZinc's transformations to Google's OR-Tools[4], there we had to install OR-Tools for MiniZinc manually, which we did using release version 9.3.10497 (commit 49b6301).

5.2 Obtaining seeds

As discussed in a previous section (Section 3.1.1) generating new inputs is significantly harder than mutation, but with the latter one we require a diverse set of seed files. Fortunately, the CPMpy team made a lot of documentation and examples on how to model problems in their language. Ranging from easy examples to teach the language to advanced examples in order to showcase certain features. At the moment

[1] `https://github.com/CPMpy/cpmpy`
[2] `https://github.com/MiniZinc/minizinc-python`
[3] `https://www.gurobi.com/`
[4] `https://github.com/google/or-tools`

of writing most examples are found in the main branch and some extras can be found on the "csplib" branch[5] waiting to be merged with the main branch. We downloaded a copy of those branches on Tuesday 27th of September to be used as future seed files.

A second source of seeds files came from Hakan Kjellerstrand a retired software developer and independent researcher from Sweden which was found while reading "Model-Based Algorithm Configuration with Adaptive Capping and Prior Distributions" [10]. Mr. Kjellerstrand has a big repository[6] full of problem models which he solves in multiple ways, including CPMpy. We obtained a copy of all his CPMpy examples on Tuesday 27th of September to top off our collection of future seed files.

After that we ran all seeds to test that the non-modified seeds do not crash on their own and noticed that most examples ran in less than a minute. The handful of examples that did run long were left out or were simplified to gain a speed up while solving them. Knowing that all seeds are capable of being run in a minute helps us avoid the halting problem. If a modified seed starts running significantly longer than a minute we can start investigating that seed for a potential bug. A final change we made to the future seed files is extracting the model from each file found on the repositories. We did this for a couple of reasons: some files had a loop around the solve instruction combined with small changes or had multiple problems in one file, this gave us a separate model for each found problem. In order to extract these constraints, we temporarily modified CPMpy to extract the created model, variables and constraints included, each time solve was called, this resulted in over nine thousand problem models which we will use as our seed files.

5.3 Modifying STORM into CTORM

Our first technique of finding bugs is heavily based on STORM which we shortly discussed before in Section 3.2.1. Instead of searching for SMT bugs, here we want to be able to find bugs in constraint programming languages and specifically in CPMpy. We downloaded STORM from the repository[7] on Tuesday 27th September. The original plan was to convert our seeds to FlatZinc using the MiniZinc API provided by CPMpy to then convert that to SMT-LIB [13] using Miquel Bofill et al.'s fzn2smt-tool to then be able to use STORM as it was built originally. Unfortunately (and a bit predictable), this way of working did not work out. On top of fzn2smt being more than a decade old, the multiple transformation layers that could introduce conversion bugs and the unclear way back from SMT-LIB prevented this path from being investigated by us.

Therefore, we decided to refactor STORM to fit CPMpy and name the technique CTORM for CPMpy-STORM. To change STORM into CTORM, we needed to rewrite the detection, labeling and construction of (sub)constraints, this refactoring did come with some downsides, some features of STORM no longer work such as

[5] `https://github.com/CPMpy/cpmpy/tree/csplib`

[6] `https://github.com/hakank/hakank/tree/master/cpmpy`

[7] `https://github.com/Practical-Formal-Methods/storm`

incremental solving or the input obfuscation that was built-in. A bigger downside came with the refactoring of the negation function of STORM, as CPMpy is still in active development and the negation not always being implemented already, this was felt while trying to negate global constraints. I.e., when trying to invert (sub)constraints which include `alldifferent([var1, var2, var3])` using CPMpy, it crashed, this is of course a bug (more specifically not yet implemented) in CPMpy but used by the fuzzer. So here we had the choice of adding the missing negation of global constraints to CPMpy or to limit our fuzzer to not use the missing features. We choose to limit the fuzzer, since we are trying to detect bugs in CPMpy with different tools and extending the language ourselves goes out of scope of this thesis.

We gave only non-flattened inputs to this solver since both STORM and CTORM used a recursive process to get all subformulas because we wanted to change as little as possible to the inner workings of CTORM compared to STORM. The flattening process in CPMpy is used to convert the potential tree-like constraint structures to a flattened list of constraints which the solver can handle. In order to get those non-flattened inputs, we hijacked the flatten process of CPMpy to return all subformulas before returning the flattened constraints, this gave access to the more convoluted subformulas to use in the next steps of CTORM. This flattening process was done before any modifications were made, so in the eyes of the fuzzer it got flattened seeds but with the knowledge of some more complex constraints just like STORM and CTORM does. For each input CTORM combines 100 new constraints built from the existing constraints, this is repeated to create a hundred models to then check if the result matched with the original output in CPMpy.

5.4 Metamorphic testing

While CTORM was quite autonomous, metamorphic testing did take one step back to manual work, as this technique requires some metamorphic transformations, these transformations take a (or multiple) constraint(s) of our seed problems and change them repeatedly while keeping the (un)satisfiability the same. To then test if the original seed problem gives the same result as our modified problem, as discussed in Subsection 3.4.2.

With papers such as [1, 57, 64] and others giving us inspiration, we came up with but not limited to the following 30 metamorphic relations. Replacing global constraints like `alldifferent([var1, var2, var3])` to their decomposition `var1!=var2 and var1!=var3 and var2!=var3`. Adding futile variables to global constraints such as `allequal([var4, var5])` by copying variables which did not limit (or restrict) the existing solution-space. We did this too for other more basic operations such as “and”, “or”, “xor”, “->” (implication), all forms of comparisons, min, max and others. We also included metamorphic relations proposed by the authors of “Validating SMT Solvers via Semantic Fusion” [64], those being semantic fusion for addition, subtraction, multiplication, and, or, xor and the comparisons. All analog to the example given in Subsection 3.2.1.

We also linked multiple (sub)constraints of the problem to each other and replaced

comparisons by other equivalent comparisons. Lastly, we also added new constraints which were independent of the original problem only to get in the way of the seed problem or be used in other metamorphic relations.

All these metamorphic relations individually were quite simple and should be handled easily by the flattening process, other CPMpy processes or by the solvers, but by combining multiple relations at random we were able to create more complex constraints that were not always handled correctly. Finally, we should note that while finishing this thesis Jo Devriendt found a bug within the metamorphic tester that made cyclic expressions possible which will crash CPMpy, which was swiftly fixed in the tester. More information about this bug can be found in the CPMpy bug report number 163.

5.5 Differential testing

With differential testing we stepped a bit further away from the fuzzing world since we did not change the input files. With this last technique we put the (sub)solvers against each other, if solver A said that the problem is unsatisfiable and all other solvers said that the same problem was satisfiable we can say that we found a bug and that the bug was most likely to be in solver A, which differs from the previous techniques where we knew the correct solution in advance.

The way this tester was written is quite simple, we tested a given input on multiple solvers and searched if there was any difference in outputs or on the number of solutions provided in the results. However, we discovered that only two solvers, namely “ortools” and “gurobi”-solvers, are able to search all solutions for problem models that contain global constraints. More solvers are available to find all solutions for SAT problems but the main objective of this thesis is to find CP-related bugs. The limitation to only two solvers results in only being able to compare two different implementations and has a risk of overlapping bugs. Preferably, we would have three or more solvers to be able to compare between them and also be able to automatically show us which of the solvers is likely to be wrong. In the future more solvers will be available within CPMpy with the capability of finding all solutions, but right now these tests are performed with two solvers. Small note, a limit of 100 solutions was put in place, otherwise finding solutions would take an unreasonable amount of time.

When we look at only searching for one solution for a given example, we had more than enough solvers to compare between. Most of the times 14 solvers were compared and we never got lower than 6 solvers, this variation is due to the features used in the given problem the amount of solvers changed. If a problem happens to not use any global constraints the SAT-solvers may be able to solve them as well allowing us to compare between up to 36 solvers. Given that we only search for a single solution we are limited in what we can compare. For example, if solver A says that it found a satisfiable problem with the solution having values A’ and solver B says the same but with different values B’ (with $A \neq B$) the only thing we can compare is that both solvers output the same “satisfiable”. Since both solution A’ and B’ can be different solutions to the same problem.

5.6 Detecting the cause of the bug

As described in Chapter 4 finding a bug is one part of the problem, the other is finding which part of the inputs causes the bug. Submitting a complex input would result in a lot of work for the development team in order to find the cause. To avoid this situation, we used a Minimal Unsatisfiable Subset finder created by the CPMpy-team. The first version we used can be found in the advanced folder[8] of the examples of CPMpy version 0.9.9. It was limited to finding MUS with the OR-Tools solver only, which caused us to write our own programs to deobfuscate the inputs based on this MUS-finder.

We created a program to simplify inputs as described in Subsection 4.1.1 and a program to isolate inputs as detailed in Subsection 4.1.2. Nevertheless, we ended up only using the simplification technique due to not wanting to report a crucial part of the bug in one issue and reporting the difference between two inputs in the next issue. The already mentioned time penalty was not noticeable due to the inputs not being enormous. Both the programs we created did not give a minimal input back but did minimize the number of constraints, the difference being that some constraints could contain non-crucial parts of the bug. For example, a constraint "[15][variable1](...)" would give the exact same bug as constraint "[0, 0, 0, 0, 0, 0, 0, 0, 0, ... 0, 15][variable1](...)" but be clearer for the developer.

To achieve a minimal input, we were planning to update our previously created programs, but CPMpy version 0.9.10 came out and contained a better MUS-finding program[9] suitable for finding MUS for all available solvers, which was important since not all (sub)solvers would agree on the problem being satisfiable or not.

Nevertheless letting a solver search for an unsatisfiable subset in what it thinks is a satisfiable problem does not work, as it will never find an unsatisfiable subset for obvious reasons. It was a similar case for crashes, for which we used a modified version of our simplification program. For obvious reasons the MUS-finders of CPMpy did not work for crashing programs. Our simplification program would test if the model still had a crash with some missing constraints or not and continue analog to the MUS version. It would again result in a minimal number of constraints but not in a minimal model. To get a minimal model some manual work was needed but that was not a significant amount of work.

5.7 Conclusion

In this chapter we discussed which software was used in tandem with the version's numbers and GitHub repositories where applicable. We showed how we turned CPMpy's and Hakan's examples into the seed files we then used for our three techniques. After that, we specified how we implemented our techniques to find bugs, this being the modification of STORM into CTORM, the creation of metamorphic

[8] `https://github.com/CPMpy/cpmpy/blob/master/examples/advanced/musx.py`
[9] `https://github.com/CPMpy/cpmpy/blob/master/cpmpy/tools/mus.py`

relations and the comparison of the output of (sub)solvers in the differential testing. To finally end with the tools used to minimize the input files once a bug was found.

Chapter 6

Results

In this chapter we discuss the results of all three of the techniques introduced in previous Chapter 5, we explain how we prevented frequently occurring bugs, discuss a diverse subset of found bugs with more detail. To end the chapter with a classification of all bugs and the reception of the bugs.

6.1 Running the tests

All tests were executed on an Ubuntu 20.04.5 LTS with 8GB RAM, an Intel core i5-3380M capable of 2.90GHz and a V-NAND SSD of 500GB 860 EVO Model MZ-76E500 through a SATA 3 connection. Other software versions used can be found in the first section of the implementation Chapter 5, with each technique taking around a day to three days to run more than the nine thousand seed files once. Note that processing a seed once can mean that variants of that seed were run up to 100 times depending on the technique used.

6.1.1 Preventing the same bug from occurring frequently

After initial experimentation we noticed that once a bug was found by any of our techniques, the same bug would occur frequently, but in a different seed causing our resulting logs to be cluttered with duplicate bugs. Although we were well aware of the possibility these duplicate bugs could appear, we underestimated the severity of impact on our workflow. Whereas we planned on using a reactive deduplication technique in post-processing as described in Subsection 4.3.1, after 10.000 occurrences of equivalent bugs, we changed to a preventative approach.

Once a bug was noted, we tried to prevent it, for crashes this meant adding a try catch for the specific error, while for wrongly (un)satisfiable bugs we looked at special occurrences of keywords in the constraints we knew caused bugs, to then blacklist them. For example, a bug we will discuss in the next section had a specific string of characters, " == 0 == 0". Being able to check the string of characters knowing that it resulted in wrongly unsatisfiable solutions made it possible for us to filter that bug as well.

Luckily, with these restrictions on the output logs we were able to filter out most duplicate bugs. Naturally this is not an ideal solution, but in this situation it worked well enough. The ideal solution would be that the fuzzer has knowledge of the already found bugs and rejects them. Techniques like these do exist but would bring us too far from the scope of this thesis. Unfortunately, adding strings to blacklists and restarting the tests did result in extra manual overhead.

6.2 Results: found bugs

In total we found 19 bugs, three of which were already known in one form or another as an issue. Of those 19 bugs some of them were easy fixes, some were harder and required more time to solve. Depending on which definition of a bug is used, the number of found bugs will differ, with the definition followed in this thesis, a crash, hang or wrong output, 19 bugs remain out of the 22 submitted. At the time of writing not all bugs are resolved (some are just reported days ago). Nevertheless, we are working actively with developers from the CPMpy library to resolve the open bug reports. For now let us look at what we deem to be the most interesting bugs. In order to do this, we will work with the four components we defined earlier in Subsection 2.2.3, this being the model, the transformations, the solver interface and the solvers themselves as seen in Figure 2.1.

6.2.1 Double Negation

The first bug we discovered was a bug involving a double negation, a bug where we ask CPMpy to solve the equivalent constraints “X==3” and “not(not(X==3))” at the same time. Given the domain of ‘X’ contains 3 this solution is trivial. Set variable ‘X’ equal to 3 and the problem would be satisfied. However not all CPMpy solvers did agree with this, both OR-Tools and Gurobi said that this problem was unsatisfiable through their corresponding CPMpy interface.

This was due to a process within CPMpy responsible for creating a flat normal form, as described by the documentation of CPMpy[1] not all solvers interfaced by CPMpy allow an arbitrary nesting of constraints. It is for this reason CPMpy flattens the constraints to what they call “flat normal form”. With a disclaimer that this definition does not formally exist for CP languages to their knowledge, a statement which we agree with. With this flattened form, CPMpy is able to directly call the solvers or do the last changes needed for the specific solver via the solver interface on the flattened constraints to then send it to the respective solver [28].

Within CPMpy all negations get translated to a comparison with a zero “== 0”. Making the double negation turn into a double comparison with zero which was not handled correctly in the normalizing process, causing a disappearance of a single “not”, which in turn resulted in the original constraint converted to the not equivalent constraints “X==3 and not(X==3)”. When this gets sent to OR-Tools or Gurobi they correctly answer with “unsatisfiable”. The other solvers, mainly MiniZinc’s

[1] `https://cpmpy.readthedocs.io/en/latest/behind_the_scenes.html`

subsolvers were not affected by this bug due to not using this normalizing process. Although this normalizing process was subjected to unit tests, these tests contained an incorrect output causing the bug to remain hidden, this bug was only caught using CTORM, due to its frequent use of adding negations and conjunctions. Due to the lack of metamorphic relations containing a double negation this bug was not caught using the metamorphic testing. Neither differential testing caught the bug since no seed had a double "not" in its constraints.

A showcase of this "double negation" bug can be seen in Listing 6.1, where a variable 'X' is created on line 3 with a lower bound (lb) of zero and an upper bound of 9 (ub). Then, add the constraint "X == 3" to a created model on line 5 with the "+=" and the same constraint with a double negation on the next line. Remember that CPMpy uses '∼' as a negation. We then see the different solvers solve the same model with a different exit status, unsatisfiable for OR-Tools and Gurobi and feasible for a MiniZinc subsolver. Feasible is a differentiation made by CPMpy within satisfiable with the other option being optimal both explain themselves. This bug report can be found in the GitHub repository issue number 142.

```
1  from cpmpy import *
2
3  X = intvar(lb=0, ub=9)
4  m = Model()
5  m += X == 3
6  m += ~(~(X == 3)) # double negation
7
8  m.solve(solver="gurobi")
9  print(m.status().exitstatus.name) # UNSATISFIABLE
10
11 m.solve(solver="ortools")
12 print(m.status().exitstatus.name) # UNSATISFIABLE
13
14 m.solve(solver="minizinc:chuffed")
15 print(m.status().exitstatus.name) # FEASIBLE
```

Listing 6.1 – The "double negation"-bug.

The constraints of the Model can be seen in Listing 6.2, with the variable 'X' on line 2 and both constraints on line 4 and 5. After normalization we can see that we have lost a negation in Listing 6.3. The remaining negation has been put into the equation of "X == 3".

```
1 Variables:
2 X: 0..9
3 Constraints:
4 X == 3
5 X == 3 == 0 == 0
6 Objective: None
```

Listing 6.2 – The constraints of the "double negation"-bug *before* the normalization process.

```
1 Variables:
2 X: 0..9
3 Constraints:
4 X == 3
5 X != 3
6 Objective: None
```

Listing 6.3 – The resulting constraints of the "double negation"-bug *after* the normalization process.

6.2.2 Negation of global constraints

A second bug related to the use of negations were the crashes of the negated global constraints, where negations of global constraints like `not(AllDifferent(argList))` would crash with a recursion error. As the normalizing of a negated global constraint would be handled with adding a "== 0" to it. The action of adding a "== 0" to the constraint did not change anything causing the same to happen when the next normalization would be attempted on the global constraint. The solution was to negate the decomposition of the global constraint instead of negating the global constraint, which was suggested in the comments together with a commented out raising of a not-implemented error. The entire function was labeled as work in progress, but the CPMpy-team expected it to work for this use case as they used it in their reification process. The reason it worked in this process was due to a shortcut not being taken, which the negation of global constraints did do, therefore exposing the bug only in the latter case.

It was again due to the normalizing not being used for MiniZinc's subsolvers that the bug only occurred when using OR-Tools and Gurobi as solvers. Moreover, this bug was quickly found by CTORM since it uses a significant number of negations. Due to the metamorphic relation of adding "!= 0" after some constraints the metamorphic tests managed to find it as well. However, it did not get found by our differential tests as no examples negated their global constraints.

A showcase of this bug can be seen in Listing 6.4, where a variable "pos" is created on line 3 with a shape of 3 meaning that "pos" will be an array of length 3. After creating an empty model, a negation of a global constraint "AllDifferent" is added to a created model on line 5. By negating this constraint, we require the solver to find an assignment to the variables in the array "pos" where at least two have the same value. For example, array "[1 2 2]" would satisfy, but "[1 2 3]" would not. Subsequently, a MiniZinc subsolver is used to solve the problem which returns feasible on respectively line 7 and 8. However, when sending the same to Gurobi it will crash, analogous with the next line if the previous one would not have crashed the program. This bug report can be found in the GitHub repository issue number 143.

```
1  from cpmpy import *
2
3  pos = intvar(lb=0, ub=5, shape=3)
4  m = Model()
5  m += ~AllDifferent(pos)
6
7  m.solve("minizinc:chuffed")
8  print(m.status().exitstatus.name) # FEASIBLE
9
10 m.solve("gurobi") # crash
11 m.solve("ortools") # would crash as well
```

Listing 6.4 – The "negation of global constraints"-bug.

6.2.3 Power function of Gurobi

Now we have seen two bugs in the transformation part of CPMpy, which both fit in the transformation component of CPMpy. It is time to look at the solver interface and some bugs we found there. The first one was a bug where the solver Gurobi would crash if we gave a base variable in the power function which had a negative lower bound. Lower bound meaning that the variable was not permitted lower than that specific value. All other solvers would be able to solve "pow(X, 2) == 9" with the variable 'X' defined with a lower bound of -5 and a higher bound of 5. However, Gurobi did not allow this and raised an error as can be seen in Listing 6.5, this error was then not caught by CPMpy and therefore could not be turned into an error exit status as a result.

This bug was found with the CTORM implementation because there was a seed file which contained this power function with a negative lower bound in the base. However, the solver used to solve this problem was not Gurobi meaning that the bug was not discovered when the example was written. The original example can be found at the CPMpy repository's csplib examples[2]. We do not think that CTORM would have found this bug if it was not in the seed file to begin with, this because CTORM does not create new variables nor modifies any bounds of variables.

The bug was also not found using the metamorphic tests, since no test covered the power function or changed any bounds of variables, which is not a limitation of metamorphic testing but a limitation of added constraints by us. Additionally, since the problem was already in the seed file to begin with, the differential testing did find the bug and logged it. This bug report can be found in the GitHub repository issue number 149.

```
1  from cpmpy import *
2
3  m = Model()
4  X = intvar(lb=-5, ub=5)
5  m += pow(X, 2) == 9
6
7  m.solve(solver="gurobi") # GurobiError
```

Listing 6.5 – The "power function of Gurobi"-bug.

6.2.4 Wrong bound value Error

A second bug we found in the solver interface was a missing check on the variable type, this time not in Gurobi but in the PySAT implementation. When checking if a sum of boolean variables matches a specific variable and that variable happens to be an integer instead of a boolean variable causing an error. In this specific case it was a follow-up function still within CPMpy that crashed, this with wrong bounds since it expected a bound of only two possibilities, a boolean variable, but got a larger bound. In all other places we could find a check with an error that would

[2] https://github.com/CPMpy/cpmpy/blob/b60310d7962bc7631bcf0b9024140e47c1fb302e/examples/csplib/prob005_auto_correlation.py

be reported. However, on this spot it was missed, which was quickly patched after reporting it.

Although, CTORM was run with PySAT's subsolvers, it did not find this bug simply due to a check of (un)satisfiability of the original problem at the start of the program. The technique would assume that the original seed was faulty to start with and continue with another solver or another seed. The same happened with the metamorphic tests where we needed to know the (un)satisfiability of the model before the changes, where it would crash again on the original model. Since those crashes were not logged in both techniques, we did not find it with these techniques. The bug did get discovered with the differential tester where each crash did get logged on top of all differences, this bug report can be found in the GitHub repository issue number 150.

6.2.5 Naming variables

Now we have seen bugs occur in both the transformations and the solver interface. Let us look at a bug we have found in the model component of CPMpy.

CPMpy has multiple features like importing, exporting models, adding names variables (not to be confused with the local variables as seen on line 12 in Listing 2.2) and more. The adding of the name is to make sure that after an export and import the given variable names are still remembered among other reasons, like to be able to give the solver the variable names. When the programmer does not give a name to a variable as we did on line 12 in Listing 2.2 with the missing `name="` attribute in the `intvar()` function. Then CPMpy adds a name to the variable without telling the programmer. These variables start with "BV" for boolean variables and "IV" with integer variables and each get appended with their respective incrementing number to prevent similar names, this is because reusing variable names is dangerous as the solvers use this name to differentiate variables from each other.

This brings us to the bug; it occurs when importing a model with automatic naming where those counters did not get updated. Meaning that when a new variable was created with automatic naming it would have an overlapping variable name with a variable that was previously imported. When a solver was then called it would treat both variables as the same resulting in potential wrongly unsatisfiable solutions. The use case is a bit farther from the normal use case a programmer would go through. Nevertheless, this was not considered a misuse of CPMpy according to the developers and at the time of writing a pull request was proposed in which the import function got extended to check the highest occurrence of the boolean and integer counter, this highest occurrence will then be used for the counters of new variables.

An almost related bug is in the naming of variables, when creating them starting with non-alphanumeric symbols like '+', '%' or others some solvers would crash. Most solvers would happily solve with these names, but MiniZinc crashed with a syntax error when handling the input. Due to the transformation of our model to the text-based Zinc for the subsolver, it can no longer differentiate between the variable name and the code. It therefore crashed when seeing anything that could

be interpreted differently than a variable name. MiniZinc does state that identifiers are not allowed to contain special characters, which other solvers and CPMpy do allow. A solution is still being discussed at the time of writing.

In Listing 6.6 this bug of non-alphanumeric symbols is showcased, with a variable 'i' being declared on line 3 with a lower and higher bound respectively 0 and 5. To then define a name manually and name it '+', this in contrast with the previous listings where CPMpy used automatically naming of the variables. Due to this non-alphanumeric naming of variables MiniZinc will crash with a syntax error on line 13 while it solved the constraint fine on line 7. Lines 10 and 11 are only added to show what the MiniZinc solver receives which is visible in Listing 6.7.

```
1  from cpmpy import *
2
3  i = intvar(lb=0, ub=5, name="+")
4  m = Model()
5  m += i > 0
6
7  m.solve(solver="ortools")
8  print(m.sstatus().exitstatus.name) # OPTIMAL
9
10 s = SolverLookup.get("minizinc", m)
11 print("".join(map(str, s.mzn_model._code_fragments)))
12
13 m.solve(solver="minizinc:chuffed") # crash by syntax error
```

Listing 6.6 – A bug showcasing that the naming of CPMpy's variables is less strict than MiniZinc's naming.

In this second listing (Listing 6.7) we can clearly see that the '+' of line 4 and 5 are out of place syntactically, which in turn causes the error.

```
1  % Generated by CPMpy
2  include "globals.mzn";
3
4  var 3..6: +int;
5  constraint (+int) > 4;
```

Listing 6.7 – The resulting Zinc code from Listing 6.6 used by Minzinc.

Both of these bugs were not caught by CTORM nor the differential testing, since they do not create new variables. However, the bugs did get caught by the metamorphic testing, the first bug was caught because we imported a seed file where automatic naming was done after which we created a variable too with this process, resulting in the bug. The second one is a bit more embarrassing to write down, as we created a bug in the metamorphic tester which resulted in the (unintentionally) creation of variables starting with a '+'. Our own bug caused us to find a bug in CPMpy. We still label it a caught bug because the automatic bug catcher did find it, although only by a fault we made. These bug reports can be found in the GitHub repository issue number 158 and 162 respectively.

6.2.6 MiniZinc returning zero

Our last bug we will discuss in detail was a bug we found with a solver themselves, namely with some MiniZinc subsolver, while solving certain problems with MiniZinc's

Table 6.1 – Table discussing in which CPMpy components the bugs were found, with 4 bugs in the model, 7 bugs in the transformations, 7 bugs in the solver interface and one solver bug was found.

BugNr	Bug description	Place of the bug
142	double negation gives unsat	Transformations
143	negating global constraints crashes	Transformations
145	solvers lookup crashes	Model
149	power function with negative lower bound crashes	Solver interface
150	wrong bound causes a crash	Solver interface
152	boolean variable does not support implies	Model
153	Gurobi does not run and gave the wrong nr of sol	Solver interface
154	JSON Decoder error	Solver interface
155	list has no shape	Solver interface
156	MiniZinc returns zero causes a crash	Solver
157	circuit of one element crashes	Transformations
158	identical variable name can cause wrongly unsat	Model
159	unhandled Gurobi exit status 9	Solver interface
161	two separate references for the same variable	Model
162	CPMpy is looser with variable names than MiniZinc	Solver interface
164	malloc() failure due to unset bounds	Transformations
165	memory violation segmentation fault	Transformations
168	unsatisfiable Gurobi	Transformations
170	unsatisfiable due to flattening	Transformations

subsolvers Gecode and others would sometimes crash with the error that it stopped without output. After reporting it turned out to be a known bug in the MiniZinc Python repository for Windows operating systems and was fixable with setting some path variables correctly, which CPMpy may solve by adding a warning when this happens or by documenting it. Given that this problem is an installation problem, all techniques were able to find the bug. This bug report can be found in the GitHub repository issue number 156.

6.3 Classifications

Now that we have seen in depth explanations of some bugs, let us give an overview of all found bugs by classifying them based on place, type of the bugs, which solver caused the bugs and which technique found the bugs. The bug number refers to the issue number on GitHub and is a hyperlink to that bug, the second column is a short description of the bug and then the table specific classification follows.

As can be seen in Table 6.1 the cause of which component failed is well spread out within CPMpy. With 4 bugs in the model, 7 bugs in the general transformations, 7 bugs in the solver interface and one solver related bug. The one and only solver related bug we found was the one we discussed in Subsection 6.2.6, which was

Table 6.2 – Table discussing what types of faults were caused by the bugs.

BugNr	Bug description	Type of fault
142	double negation gives unsat	wrongly unsat
143	negating global constraints crashes	crash
145	solvers lookup crashes	crash
149	power function with negative lower bound crashes	crash
150	wrong bound causes a crash	crash
152	boolean variable does not support implies	crash
153	Gurobi does not run and gave the wrong Nr of sol	wrong Nr of sol
154	JSON Decoder error	crash
155	list has no shape	crash
156	MiniZinc returns zero causes a crash	crash
157	circuit of one element crashes	crash
158	identical variable name can cause wrongly unsat	wrongly unsat
159	unhandled Gurobi exit status 9	crash
161	two separate references for the same variable	wrongly unsat
162	CPMpy is looser with variable names than MiniZinc	crash
164	malloc() failure due to unset bounds	crash
165	memory violation segmentation fault	crash
168	wrongly unsatisfiable Gurobi	wrongly unsat
170	wrongly (un)satisfiable due to flattening	wrongly (un)sat

already known by MiniZinc. We would have hoped to find more bugs in the solvers themselves and it was our aim with this thesis. However, either these techniques are not sufficient or most bugs are already found before release or are already reported and solved.

If we look at the reported issues within the GitHub repository of Google's OR-Tools, we find no significant bugs towards the (un)satisfiability or any wrong output by the solver, which makes us speculate that Google does extensive testing on that front or even has used the techniques used in this thesis. The last one is likely as Google created multiple own fuzzers, which we discussed in Subsection 3.2. Extensive testing is most likely also done by other solvers since they would probably lose reputation if their solver would be proven to not produce the correct result.

Like the authors of STORM, we focused with our techniques on the critical faults, this being the wrongly satisfiable, the wrongly unsatisfiable and the wrong number of solutions. Since these critical bugs are harder to detect for the final user than a crash, timeout or other bug. Out of the 19 bugs found 6 of them fall in our category of critical while the other 13 where all crashes as can be seen in Table 6.2. Most of those 6 critical bugs were situations where the solver wrongly outputted that a solution was unsatisfiable and there was only one bug where we could find both a wrongly satisfiable and wrongly unsatisfiable solution.

When looking at Table 6.3 we can see which bug was caused by which solver or if it was a solver independent bug, where we see 5 bugs unrelated to any solver,

Table 6.3 – Table discussing which bugs were caused by which solvers or if it was a solver independent bug.

BugNr	Bug description	Which solver caused it
142	double negation gives unsat	OR-Tools and Gurobi
143	negating global constraints crashes	OR-Tools and Gurobi
145	solvers lookup crashes	solver independent
149	power function with negative lower bound crashes	Gurobi
150	wrong bound causes a crash	all PySAT subsolvers
152	boolean variable does not support implies	solver independent
153	Gurobi does not run and gave the wrong nr of sol	Gurobi
154	JSON Decoder error	MiniZinc's subsolver osicbc
155	list has no shape	Gurobi
156	MiniZinc returns zero causes a crash	multiple MiniZinc subsolvers
157	circuit of one element crashes	solver independent
158	identical variable name can cause wrongly unsat	solver independent
159	unhandled Gurobi exit status 9	Gurobi
161	two separate references for the same variable	solver independent
162	CPMpy is looser with variable names than MiniZinc	all MiniZinc subsolvers
164	malloc() failure due to unset bounds	multiple MiniZinc subsolvers
165	memory violation segmentation fault	multiple MiniZinc subsolvers
168	wrongly unsatisfiable Gurobi	Gurobi
170	wrongly (un)satisfiable due to flattening	OR-Tools and Gurobi

that OR-tools only occurred together with Gurobi and that OR-Tools and Gurobi didn't share any bugs found with MiniZinc or PySAT. Mainly because OR-Tools and Gurobi share more transformation code than any other solver pair. We also see that Gurobi occurs the most among our bugs, this often due to edge cases on Gurobi's implementation.

The last Table 6.4 shows which technique found which bug. CTORM found 10 bugs, metamorphic testing found the most bugs at 13 and differential testing found 11 out of the 19 found bugs. The results show that none of the techniques are perfect on their own and in order to find all bugs, a combination of techniques would be needed. As the industry's quality assurance processes do for finding bugs in other software packages, where they use a combination of tools as discussed in Section 1.1, with a side note that metamorphic testing could come close if more work was put in creating metamorphic relations. Although, this does require creativity and manual labor instead of the other automated techniques.

6.4 Reception to the bugs

As mentioned in Section 3.6 there are multiple views on automated bug catching. We could have reported all our found bugs on the issue page of GitHub without further context, which would have meant more work for the CPMpy-team and then we

Table 6.4 – Table discussing which techniques found which bugs. CTORM found 10 bugs, metamorphic testing found the most bugs at 13 and differential testing found 11 out of the 19 found bugs.

BugNr	Bug description	Bug found by		
142	double negation gives unsat	ctorm		
143	negating global constraints crashes	ctorm	meta	
145	solvers lookup crashes			diff
149	power function with negative lower bound crashes	ctorm		diff
150	wrong bound causes a crash			diff
152	boolean variable does not support implies		meta	diff
153	Gurobi does not run and gave the wrong nr of sol			diff
154	JSON Decoder error	ctorm	meta	diff
155	list has no shape	ctorm	meta	diff
156	MiniZinc returns zero causes a crash	ctorm	meta	diff
157	circuit of one element crashes		meta	
158	identical variable name can cause wrongly unsat		meta	
159	unhandled Gurobi exit status 9	ctorm		diff
161	two separate references for the same variable	ctorm	meta	
162	CPMpy is looser with variable names than MiniZinc		meta	
164	malloc() failure due to unset bounds		meta	
165	memory violation segmentation fault		meta	diff
168	wrongly unsatisfiable Gurobi	ctorm	meta	diff
170	wrongly (un)satisfiable due to flattening	ctorm	meta	

could perhaps have seen some negative opinions on fuzzing. Although we submitted 22 bugs or questions within a time span of 2 weeks with deobfuscating the inputs and some explanation of what happened to get the bug, we saw a grateful welcome. Similar to what was described in the second part of Section 3.6. For example, the "double negation"-bug[3] was called a "serious bug and a great find" and the "negation of global constraints"-bug[4] was described as an unexpected bug and "another great find".

6.5 Conclusion

In this chapter we have seen that the techniques frequently output already found bugs, since none of the techniques have knowledge of already found bugs. However, after filtering previously found bugs the techniques performed well and found 19 bugs in CPMpy. We have seen some bugs in detail with most of them already fixed, due to being easily fixable. Most bugs were related to the CPMpy code and only one was responsible for an external solver bug. We suspect this lack of external solver bugs to be caused by well written and tested solvers. We have shown that we found

[3] `https://github.com/CPMpy/cpmpy/issues/142`
[4] `https://github.com/CPMpy/cpmpy/issues/143`

crashes up to critical bugs like wrongly (un)satisfiable solutions and wrong number of solutions with over a variety of solvers. Of all the techniques used, metamorphic testing came just above the other two techniques with 13 found bugs while CTORM found 10 and the differential testing found 11 bugs out of 19. Finally, we noted the grateful welcome of bugs by the CPMpy-team.

Chapter 7

Conclusion and Future Work

In this chapter we will conclude the thesis by discussing the achievements, the limitations of the techniques used and end with possible future work.

7.1 Achievements

This thesis achieved to find numerous bugs within CPMpy with the use of multiple techniques. As discussed in Section 6.3, some bugs were found by multiple techniques. Out of the 19 bugs found 10 were found by CTORM, 13 by metamorphic and 11 by differential testing, these bugs were mostly found well spread within CPMpy, but very little were found in the solver themselves. A third of the time the bugs resulted in a critical bug, this being wrongly (un)satisfiable or the wrong amount of solutions presented. The other two-thirds of the bugs resulted in a crash. Often caused by smaller edge cases where CPMpy did not follow the specific solver's specifications, for example the power bug of Gurobi in Section 6.2.3. None of the techniques got a perfect score meaning that when looking for all bugs a combination of tools will be needed as clarified in Subsection 6.3. We have also shown that there are multiple techniques to find critical bugs that are not easily spotted otherwise and we have multiple CPMpy specific fuzzing tools.

The techniques almost achieved semi-automatic testing with the corresponding advantages of ease of use, time savings and more while extensive testing. Both CTORM and differential testing have potential to become fully automatic testing tools after fixing the repeated logging of already found bugs.

7.2 Limitations

This problem of frequently finding the same bug is a problem, since it clogs the output with repeated or similar bugs, making it harder to find new bugs. The first solution proposed in this thesis was to deduplicate bugs with the technique seen in Subsection 4.3.1. However, during testing a preventative approach seemed favorable compared to the reactive approach of filtering out known bugs. Although the filters currently in use work, they require the developer to run a tester, see what the results

are and add a try catch for the most occurring bugs. Ideally, this manual creation of filters is done automatically by adding the knowledge of the already found bugs to the tester.

Another limitation to the testers is specifically with the metamorphic testing, in this technique metamorphic relations need to be manually created, these relations are then used to change constraints to an equivalent but different constraint. With more particular relations the technique would have been able to find at minimum the "double negation"-bug and the power bug of Gurobi as well. However, after implementing 30 metamorphic relations creativity for new relations starts to dwindle down. A second limitation will be that these relations will need to be updated or added after each new addition to CPMpy, which results in more work for the developer and another step away from automation.

While working with seeds is better than generating inputs themselves as discussed in Subsection 3.1.1, it did bring limitations with it. Those limitations being the availability, complexity and diversity of the seeds. For example, the bug with the negation of global constraints was not found using differential testing since none of the over nine thousand seed files had any negation of a global constraint. This was a limitation created by using seed files that originated from examples of CPMpy.

7.3 Future work

With new code new bugs will appear so the work of a developer will never be done when it comes to finding bugs. Rerunning the techniques could result in even more interesting bugs. On top of solving the limitations specified in the previous Section 7.2, a look at testing the configuration space would be an interesting addition to the performed study, this testing of configuration space was briefly mentioned in Section 3.2.1 where the authors of "Fuzzing SMT solvers via two-dimensional input space exploration" [65] also fuzz test the configuration space of the PUT. For example, there could be bugs that only occur when certain optimizations are turned on or off like: dynamic symmetry breaking or others. Implementing this fuzz testing of the configuration space would definitely be possible within CPMpy as it has direct access to the solvers. Finally, an extension to other languages would be possible as a future work as well.

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] *American Fuzzy Lop Rabbit – Complete Guide*. English. animalcorner. URL: `https://animalcorner.org/wp-content/uploads/2020/11/American-Fuzzy-Lop-Rabbit.png` (visited on 09/21/2022).

[3] 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.

[4] Haniel Barbosa et al. “cvc5: a versatile and industrial-strength SMT solver”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pp. 415–442.

[5] 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.

[6] 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.

[7] Clark Barrett and Cesare Tinelli. “Cvc3”. In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 298–302.

[8] Clark Barrett et al. “CVC5 at the SMT Competition 2021”. In: ().

[9] 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.

[10] Ignace Bleukx et al. “Model-based algorithm configuration with adaptive capping and prior distributions”. In: *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2022, pp. 64–73.

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

[12] 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.

[13] Miquel Bofill, Josep Suy, and Mateu Villaret. "A system for solving constraint satisfaction problems with SMT". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2010, pp. 300–305.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Marc Denecker. *Modelling of Complex Systems*. English. Cursusdienst VTK Ondersteuning vzw, Jan. 3, 2022.

[19] David Detlefs, Greg Nelson, and James B Saxe. "Simplify: a theorem prover for program checking". In: *Journal of the ACM (JACM)* 52.3 (2005), pp. 365–473.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] Alan M Frisch et al. "Essence: A constraint language for specifying combinatorial problems". In: *Constraints* 13.3 (2008), pp. 268–306.

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

[27] Tias Guns. *CPMpy: Constraint Programming and Modeling in Python*. English. URL: `https://cpmpy.readthedocs.io/en/latest/index.html` (visited on 09/15/2021).

[28] Tias Guns. *CPMpy: Constraint Programming and Modeling library in Python, based on numpy, with direct solver access.* English. 2017. URL: `https://github.com/CPMpy/` (visited on 09/15/2021).

[29] 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`.

[30] 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.

[31] 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.

[32] 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.

[33] Hadi Katebi, Karem A Sakallah, and Joao P Marques-Silva. “Empirical study of the anatomy of modern SAT solvers”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2011, pp. 343–356.

[34] Ruben Kindt, Patrick Vandewalle, and Nils Deslé. “How to get a consistent editing result independent from frame rate while grinding dental surface scans”. English. Thesis. KU Leuven, July 2020.

[35] 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.

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

[37] Yuwei Li et al. “V-fuzz: Vulnerability-oriented evolutionary fuzzing”. In: *arXiv preprint arXiv:1901.01142* (2019).

[38] 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.

[39] 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.

[40] 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.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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).

[48] 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`.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] Peter J Stuckey. "Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving". In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer. 2010, pp. 5–9.

[55] Peter J Stuckey, Ralph Becket, and Julien Fischer. "Philosophy of the MiniZinc challenge". In: *Constraints* 15.3 (2010), pp. 307–316.

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

[57] 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.

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

[59] 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.

[60] 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.

[61] *Website/aflppbg.svg at master AFLplusplus/Website GitHub*. English. GitHub-AFLplusplus. URL: https://github.com/AFLplusplus/Website/blob/master/static/aflpp_bg.png (visited on 12/02/2022).

[62] Wikipedia. *Constraint programming*. English. Wikipedia. Sept. 13, 2022. URL: https://en.wikipedia.org/wiki/Constraint_programming (visited on 09/13/2022).

[63] Wikipedia. *Satisfiability modulo theories*. English. Wikipedia. Sept. 11, 2022. URL: https://en.wikipedia.org/wiki/Satisfiability_modulo_theories (visited on 09/11/2022).

[64] 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.

[65] 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.

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

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

[68] 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.