The best master's thesis ever

ing. Ruben Kindt

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

Promotor:
Prof. dr. Tias Guns

Begeleider:
Ir. Ignace Bleukx

Academiejaar 2022 – 2023

Thesis Title, does not have to be a '?'

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 my self to start working on it. But 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 many other CPMpy bug solvers, Hakan Kjellerstrand for publishing significant amount examples which we used as seeds, friends for proofreading and family for the support during my further studies.

proofread

ing. Ruben Kindt

Todo list

remove todo list

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 to find bugs in constraint programming languages. The first technique builds further on an existing paper to test 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.

results

abstract

proofread

add keywords

Samenvatting

samenvatting

State of the art 'in deze masterproef stellen wij een vergelijkende studie voor om auto bugs te vinden in cpl met als voorbeeld taal de CPMpy written by Tia's Guns at al. Voordelen automatisch bugs (Tijdwinst, Safer programma)

proofread

List of Figures and Tables

List of Figures

List of Tables

Listings

List of Abbreviations and Symbols

Abbreviations

CI/CD	Continuous Integration and Continuous Deployment, a pipeline for newly written code to repeatably be build, test, release, deploy and more.
CP	Constrain Programming language sometimes also referred to as CPL
CPL	Constrain Programming Language also referred to as CP
CNF	Conjunctive Normal Form, which is a boolean formula written using conjunctions of distinctions.
CSP	Constraint Satisfaction Problem is a problem with constraints and variable with a specific domain e.g., boolean, finite and others.
CPMpy	Constraint Programming and Modeling language for Python.
CVC	Cooperating Validity Checker a popular SMT-theorem prover
PUT	Program Under Test, the piece of code, application of program that is tested on for potential bugs.
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, theory where decision variables are allowed to be integers
MUS	Minimal Unsatisfiable Subset, the smallest subset possible that is not satisfiable
SMT	Satisfiability Modulo Theory
SUT	Software under Test, analogue to PUT

Symbols

$\sim$	negation used by CPMpy
&	and used by CPMpy
$\vert$	or used by CPMpy
$\neg$	logical negation
$\wedge$	logical and
$\vee$	logical or

Chapter 1

Introduction

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

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

, as quoted from [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 this on multiple different ways: code review, manual testing or automated testing. All these techniques could exist out of 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. Other cases dynamic testing may be better, image 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 fit 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), this is where 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., does it crash, give wrong results or other unwanted actions) the fuzzer can automatically detect bugs.

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

1.2 Fuzzing and security

Fuzzing is a novel way for attempting 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 problem or not. But 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 will need to invest in prevention and (early) detection. Which 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, Log4Shell, 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 Yuwei Li et al. [37] do this with their Vulnerability-Oriented Evolutionary Fuzzing tool.

1.3 Goals

With this thesis we aim to be able to compare multiple know 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 are not always clear that they occurred. The techniques we will use will be a modified fuzzer originally used on SMT problems, a second will be apply satisfiable preserving changes to known inputs and test that the original input matches the known input. Our last technique will be taking advantage to the fact that CPMpy has a big library of solvers, which we will use to check that all the solvers agree on the solutions.

1.4 Research questions

As the title of the thesis already may have spoiled it, we are trying out multiple fuzzing techniques out 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 constrain programming languages. We put forward two regions of research questions we will to focus on.

1.4.1 Problem statement

As described in at 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. Try differential testing between the multiple solvers and out last technique is the use of metamorphic testing. Resulting in the following questions:
Research question 1: What fuzzing technique will find the most bugs?
Research question 2: What fuzzing technique will find the most critical bugs?
Research question 3: 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.
Research question 4: How many (critical) bugs can we find?
Research question 5: What are the causes of the bugs?

1.4.4 Not focused: efficiency and others

A keen reader may wonder why we do not focus on efficiency, this would result in more bugs being caught in a smaller timeframe. While perfectly valid to investigate we believe that discovering which techniques works best for CP' has a higher value than on top of the techniques investigated being able to run automatically. The efficiency in CP has already significant research and literature on how to optimize the solvers, which take the most amount execution time compared to the testers.

1.5 Modus operandi

This thesis will create 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 the found bugs and evaluate and evaluated the found bugs for each technique. To finally compare the results and see which techniques are able to find bugs in constraint programming languages.

optional: section, written out text what each chapter is about

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 and the paper [24] from 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. But we do not live in that type of world and problems need to be converted or split up into a representation that the solver can understand. This is where constraints come into the picture, constraints are mathematical, logical or relational connections put on or between variables to form a model that hopefully satisfy all constraints after solving. This is sometimes combined with a final objective function 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 the 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 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].

Our last field, is the satisfiability modulo theory (SMT) which builds further on SAT. This by including lists, arrays, strings, real numbers, integers and other more complex data structures and types. With SMT the focus lays more on static checking and program verification [5, 49].

2.2 Constraint programming

As mentioned before CP's are well versatile in the solving of constraints and especially when it comes to planning and scheduling. This by their efficient constraint propagation, backtracking and the linking of related constraints [62]. Originally CP's can be linked back to constraint logic programming (CLP) where programming languages (mostly logic programming languages) where 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 the category.

To further introduces you to constraint programming we will show a popular puzzle in the CP field "send more money". which is a logic puzzle made by Henry Dudeney and published in Strand Magazine's July 1924 edition [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 go as follows: all different characters should have a different digit, any of the starting letter of each word cannot be zero and after replacing all characters with their corresponding digits the sum should be correct.

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

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 this problems using constraints so that a solver can find a solution.

On line 1 you can see the importing of the global constraints, which CPs are known for. 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 functions 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 pretty-print from line 19 to 22. We will not be

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

```
include "globals.mzn";
var 1..9: S; % Since 'S' is a starting character it is limited from 1 to 9.
var 0..9: E; % Other characters are limited from 0 to 9.
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint % The sum must hold.
              1000 * S + 100 * E + 10 * N + D
            + 1000 * M + 100 * O + 10 * R + E
= 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;
output % Pretty-printing the solution.
[" \(S)\(E)\(N)\(D)\n",
"+ \(M)\(O)\(R)\(E)\n",
"= \(M)\(O)\(N)\(E)\(Y)\n"];
```

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

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 above.

The "alldifferent()" and the sum constraint also show us the power of CPs, 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, imagine the work needed for the "alldifferent()" constraint. The developer would have to go over all possible combinations, this in combination of specific smart back-end solvers for this constraint 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. which can be done with generating values within the domain of the variables and testing them, also called generate and test but obviously this is not the fastest way.

On the other hand, we have constraint optimization, which covers an even harder problem. Instead of having to check if all constraints satisfy, here 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 occurs more often than not in real life problems [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 as: local search, Tabu search, simulated annealing and more.

2.2.2 MiniZinc

A keen observer has will have noticed that we used "after a solver has found satisfiability" in the explanation of "send more money" in section 2.2. This is because MiniZinc is not a solver, it came to be from the lack of standard modeling language 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. wanted to solve, they came up with MiniZinc, which is a modeling language for CPs that is not connected to a single specific solver [50]. It originated from a modeling language focused on constraints, called Zinc [3] and as you can tell by its name MiniZinc, it is a subset of Zinc [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 to FlatZinc by combining the model, data and solver specific features. Which then can be solved by a specific solver. The list of CP solvers that support MiniZinc, at the time of writing 17 solvers have FlatZinc interfaces.

MiniZinc Challenge

On top of maintaining and improving MiniZinc the MiniZinc 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. But 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 where then able to let a SAT solver solve the problem. 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 MiniZinc challenge, it also results in more solver implementing the FlatZinc as an input and due to the competitive nature of academics brings the motivation to stride forwards according to the author of "Philosophy of the MiniZinc challenge", Julien Fischer [55].

2.2.3 CPMpy

MiniZinc is not the only one with the idea to create a modeling language for CPs, other examples are Essence with the focus on combinatorial problems for people with a background in discrete mathematics [25]. And the one with our most 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 [29]. With 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 of CPs would allow for both fields to grow closer to each other.

At the time of writing the 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 and CPMpy has support for any CP-solver that support for the 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 would have written an in-depth explanation on how CPMpy does its conversions from constraints to what is given to the solvers, but CPMpy being in development and with more changes potential coming this could become outdated quickly. We look forward to see CPMpy progress and the future paper(s) discussing it. What we can do is discuss it 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), solver interface (everything to do with restrictions the solver puts on CPMpy) and then the solver themselves (which solve 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 we included a CPMpy listing (listing 2.2.3) for the same problem as we have seen for MiniZinc.

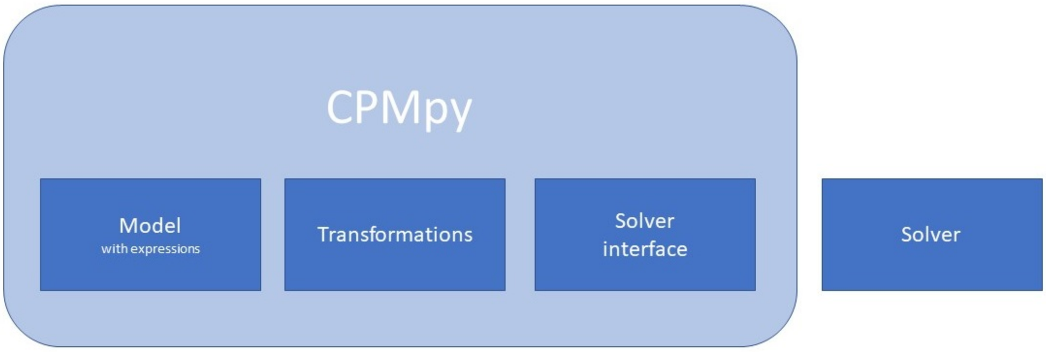

FIGURE 2.1: The four main components of CPMpy

```python
#!/usr/bin/python3
"""
Send more money in CPMpy
  SEND
+ MORE
------
 MONEY
"""
from cpmpy import *
import numpy as np

s,e,n,d,m,o,r,y = intvar(0,9, shape=8) # creating variables

model = Model() # with "+=" we can add a constraint to the model
model += sum([s,e,n,d] * np.array([1000, 100, 10, 1])) \
       + sum([m,o,r,e] * np.array([1000, 100, 10, 1])) \
      == sum([m,o,n,e,y] * np.array([10000, 1000, 100, 10, 1]))
model += AllDifferent([s,e,n,d,m,o,r,y])
model += s > 0 # in MiniZinc each variable was declared seperatly in CPMpy
model += m > 0 # we can do it in batch, resulting in these extra constraints

print(model)

# Solve and pretty-print
if model.solve():
   print("  S,E,N,D =   ", [x.value() for x in [s,e,n,d]])
   print("  M,O,R,E =   ", [x.value() for x in [m,o,r,e]])
   print("M,O,N,E,Y =",    [x.value() for x in [m,o,n,e,y]])
else:
   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's step back to boolean satisfiability problems (SAT) and as the name already spoiled it, here we are focused on checking the satisfiability of boolean formulas. SAT has been successful in the hardware design and verification. With al lot of research in improving SAT-solvers, they have become significant efficient on large problems [5]. With the efficiency improvements coming from the DPLL algorithm to conflict-driven clause learning (CDCL) move combined with the addition of non-chronological back jumps. But 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 lays in multiple fields but it is quite popular in program verification and testing. As 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 following programming languages: 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]. As like CVC4, CVC5 supports the standard SMT-LIB input format among other formats directly and has APIs for C++, Python, Java and probably more by the time you read this [8, 4].

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 becomes. 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. With this last technique the diversity of the seeding inputs does become quite important. Ideally, we would have an unlimited diverse set of inputs, but that may not always be available. A workaround to this issue could be 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, compared to only adding symbols with generational fuzzing here we also remove or change randomly selected symbols. We can create three types of inputs: non-valid, semi-valid and valid inputs. With non-valid inputs we will almost be exclusively testing the 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. And finally, 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 referrers 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. With black box fuzzing the fuzzer does not have any knowledge about the inner working of the PUT and we treat the PUT as a literal black box. We provide input and we look at what 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, as you may have predicted, 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 differentiation between black, gray and white box fuzzing is not clear cut, most people would agree that white box fuzzing has full knowledge about the PUT, including the source code, that gray box fuzzing has some knowledge about the PUT and that black box fuzzing has little to no knowledge about the PUT. Going into more detail could become controversial, all we can say is that it is no longer 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, 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 are able to be crashed with random inputs. Of the programs tested 15% to 43% of the Unix utilities crashed, 6% of the open-source GNU utilities crashed, 26% of X-Window applications crashed, 45% of Windows NT 4.0 and Windows 2000 programs crashed and 16% MacOS programs crashed.

A couple of years later, KLEE [16] was developed by Cadar et al. KLEE is a generation based white box fuzzing tool build with the idea that bugs could be on any code line and that testing should cover as code much as possible. A code coverage tool is used to test which lines of code are executed and this combined with the feedback KLEE 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, 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

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 animal-corner.org and AFL++'s website respectively [2, 61].

easily gets to 100% turns code coverage into a well-rounded measurement.

As for the more popular fuzzers, American fuzzy lop[1] (AFL), which 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. But due inactivity on Google's part the fork AFL++[2] has become more popular than the original and is maintained actively by the community [22]. Not only is it actively maintained, 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, a Go[5] focused version and is shown by Robert Heaton [30] to not be difficult to write a wrapper for it.

A potential reason to the inactivity of Google on the AFL project could be the development of both Clusterfuzz[6] and OSS-fuzz[7], a scalable fuzzing infrastructure and a combination of multiple fuzzers respectively. With the former one being used in OSS-fuzz as a back end to create a distributed execution environment. This with quite a bit of success,

> "As of 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."

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/`

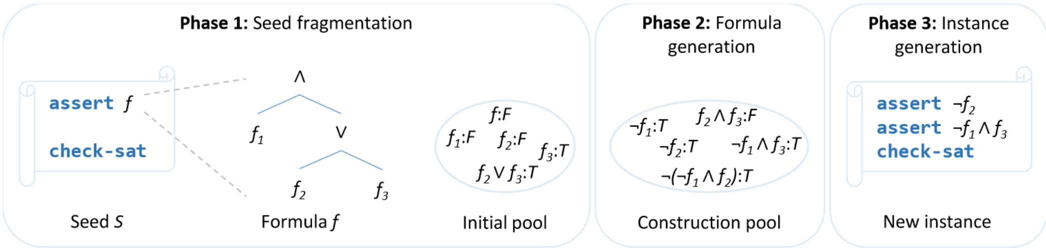

FIGURE 3.2: Overview of the three STORM phases as presented by Muhammad Numair Mansur et al. in "Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing" [40].

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. Although 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, we would like to deliberate specific fuzzers build for testing constraint programming languages (CP) and satisfiability modulo theory (SMT) solvers. One of those fuzzers is STORM which is a mutation based black box fuzzer created by Muhammad Numair Mansur et al. [40] to find critical bugs (i.e., wrongly sat or wrongly unsat) in SMT solvers. In their paper [40] they explain the inner working thoroughly, but briefly summarized STORM creates an initial pool of smaller formulas from existing formulas found in seeds, uses another solver to create models of those smaller formulas. To then construct more complex formulas with the knowledge of their ground truth, with this STORM can test the SMT solver as 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.

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

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

> 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 $\phi_1 \equiv X > 10$ and $\phi_2 \equiv Y < 9$ with the fusion function for $Z = X + Y$ would become $\phi_3 \equiv (Z - Y) > 10 \wedge (Z - X) < 9$, linking both satisfiable formulas together. For unsatisfiable formulas an extra conjunction is needed with the definition of the new variable, because a substitution could result in the loss of the unsatisfiability of the formula as mentioned in the paper [64]. The results of the paper where also significant with 45 bugs in state-of-the-art SMT solvers in Z3[11] and CVC4[12]. Dominik Winterer et al. also give multiple fusion functions 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 linking 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 17.2 to 18.8%. When knowing that SMT solver 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 a hanging inputs. With some of these bugs being less acceptable than others. For example, as Muhammad Numair Mansur et al. describes, a crash is preferred for a constraint programming language (CP) over a wrongly unsatisfied model, since there is no way for the user to know that the solver failed in that last case (except for differentiation testing, more on that later). Meaning that the user will treat the result (wrongly) as correct, 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 the fact that problems are frequently NP-hard meaning they are easy to confirm but hard to solve. For practical reasons we will be defining models that take more then 15 minutes to a timeout category and

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

wrong amount of solutions to 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. This latter one uses a single or multiple analog programs to test if the PUT gave the same output as the analog programs. As Christian Klinger et al. did in their paper [35] by applying "bugs-as-deviant-behavior" they try to find precision and soundness mistakes 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. The latter technique may therefore 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). Or when given an equivalent formula that the number of models remain the same compared to the original. This technique no longer depends on a secondary oracle but does depend on multiple executions and could miss a bug that occurs in multiple situations.

This technique has been applied in combination with differential testing to test constraint solvers with success by Akgün, Özgür et al. [1]. In which they say that this technique fits well for testing constraint solving, due to the availability of metamorphic transformations.

3.5 The oracle problem

The oracle problem describes the issue of telling 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 contains all the possible encoding of specifications like modeling languages UML, Event-B and more. Their derived test oracles classification contains all forms of knowledge obtained from documentation on how the program should work or by knowledge of previous versions of the program. The last two oracles' categories come down to the use of knowing that crashes are always unwanted and the human oracle such as crowdsourcing respectively.

3.5.1 Handling the oracle problem

Although the approach of by Bugariu and Müller in "Automatically testing string solvers" [15] falls in the first category being, a black box fuzzer, their approach is innovative. While most fuzzers either use crashes or differential testing (more on that later) to find bugs, they know the (un)satisfiability of their formulas by the way of they are constructed. For satisfiable formulas they generate trivial formulas and then by satisfiability preserving transformations increase the complexity and for unsatisfiable formulas they use $\neg$ A $\wedge$ A', with A' being an equivalent formula 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. This technique of creating formulas satisfiable by construction has also been applied to SMT solvers by Muhammad Numair Mansur et al. called STORM [40] which uses mutational input creation compared to the previous generation based techniques. In the paper the authors dissect all SMT assertions into their sub-formulas and create an initial pool. In this pool the sub-formulas are checked if they satisfy or not and with this knowledge new formulas are created for the population pool with ground truth, from this pool new theories are created and tested. This makes that STORM does not need an oracle to test the entire theory, but only the smaller sub-formulas.

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)?" and that fuzzing seems to generate an infinite number of bugs are also pet peeves of developers according to McKeeman. Although we have a bias due to writing this thesis about fuzzing, we think that those perspectives should be acknowledged in this paper. But we should also mention that this is not a single view shared by all developers from the papers [64, 65, 68] and others we see a positive response to newly found bugs by the respective developers. 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 in inputs

When we detect that the program under test (PUT) crashes, wrongly satisfied, wrongly unsatisfied, hangs or gives the wrong solution on a given input we want to know why it does that. What causes this unwanted output and on what line does the bug occurs. With crashes, a stack trace and some luck this could be easy, but when a bug causes a crash in another place the developer may need to debug deep into the code to find the bug. This with 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. We will discover this 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 can take a lot of try and error to see which variations still reveal the bug or having to walk through the execution to find the bug. Both take a while if we want to go to absolute minimal inputs, but for developers it is not needed to go to that extreme. As long as we take the bulk of the unrelated parts of inputs 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) 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 $c_{✘}$ be given such that $test(\emptyset) = ✔ \wedge test(c_{✘}) = ✘$ hold.
The goal is to find $c'_{✘} = ddmin(c_{✘})$ such that $c'_{✘} \subseteq c_{✘}$, $test(c'_{✘}) = ✘$, and $c'_{✘}$ is 1-minimal.
The *minimizing Delta Debugging algorithm ddmin(c)* is

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

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

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

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 repeatably remove parts of a failing input and check if it still fails and it often done via a "delta-debugging" algorithm, which belongs to the divide-and-conquer family of algorithms [14]. The algorithm can be seen in figure 4.1 with "$c_{✘}$" meaning the failing input to be deobfuscated, "✔" meaning that a test passed with the given input, "✘" failed with the given input, "Δ" and "∇" being a subset of the input and the complement of the former and "1-minimal" meaning that not a single character can change without the input going from failing to passing. Firstly, we start the algorithm with the input and a split "n" of 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. In case we do not find any smaller subset to continue, then we reduce the granularity of the split by 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. This input is at the same time also the shortest possible input to trigger this bug making finding the bug for the developer easier than in the original input filled with unrelated parts. An example of delta-debugging to minimize input can be found in figure 4.2. In the first two steps no removal of any part nor complement was possible therefore we reduce the granularity, after which a removal of parts 3 and 4 was found possible. To then use some previous knowledge (lines 9, 10 and 11) with 2 new tests to remove parts 5 and 6. To then decrease the granularity again, 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

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	?	$\Rightarrow$ 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	✘	$\Rightarrow$ Reduce to $c'_✘ = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	?*	Testing Δ_1, Δ_2, Δ_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	✘	$\Rightarrow$ Reduce to $c'_✘ = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	?*	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	?*	$\Rightarrow$ 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	✘	$\Rightarrow$ Reduce to $c'_✘ = \nabla_2$; 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.

input that does not show the bug. This comes with the advantage that no matter if we find the bug or not the difference will diminish, either the maximum input will shrink or the minimum input will grow. This technique brings extra complexity with the tracking of multiple inputs and bigger inputs often take longer to process, but according to Andreas Zeller et al. [67] this is the faster one to 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 of the last passed and last failed tests. And the '*' 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 subsets

For readers that are familiar with the SMT of constraint solving-world will have noticed that this techniques feels similar to a way of finding the minimal unsatisfiable

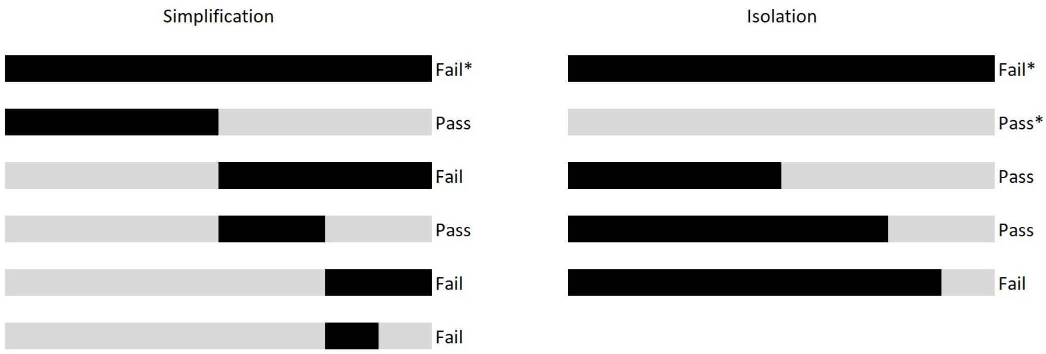

Figure 4.3: Deobfuscating inputs based on simplification (left) and isolation (right) on the same input. Figure based on an illustration found in "Why programs fail: a guide to systematic debugging" by Andreas Zeller [66].

subset (MUS), which it is in the case of a solver wrongly stating that an input is unsatisfiable. With MUS you try to find the smallest subset of formulas or constraints that will result in an unsatisfiable solution while with MSS you would be trying to find the biggest subset of formulas or constraints that would result in a satisfiable solution. Both are an iterative process and can be applied in the simplification or the isolation process. But solving which combination of formulas results in the smallest of biggest subset is a computationally intensive progress. 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 minimal input is not needed as we aim to reduces the input to help the developer find the error faster, a difference between a smaller and the absolute minimum will cause for a big computational difference in practice.

4.1.4 Alternative approach

An alternative approach compared to the already mentioned techniques is one 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 stuff to obfuscate the input. On the other hand, the chance of finding bigger bugs with multiple constraints interacting with each other will become harder. A last alternative approach would be retrying fuzzing the same seed after that seed has produced an unwanted input with adding an increasing size limitation in order to find the same bug with a 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 the inputs. The previous seen techniques will work well on the original fuzz testing by Miller et al. [45] since those random generated symbols where independent from each other. But 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-eighth of the input as the chuck sizes for the ease of the example. For performance reasons we hope we can keep our chuck sizes as big as possible to be able to discard larger unrelated parts of the inputs. But when this is not possible, we will need to decrease the granularity of the chuck sizes. For example, to be able to find the critical parts of an input of the form "XXooXooXXoo" (with 'o' being the critical parts and the 'X' being unrelated to the bug) we should always search further with same granularity while the removed parts are already removed until all options with that granularity are searched [66]. This will make sure that we eliminate all unrelated parts with the specific granularity and get "ooXoooo" instead of "ooXooXXoo".

For more complex inputs we can apply techniques seen in section 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 syntactical knowledge (the meaning of word combinations) can be used to help us in deobfuscating complex inputs. Where syntactical knowledge would help us remove the most since it is the bigger of the two.

4.2.1 Preserving satisfiability

With the techniques as 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 with the following output, then we cannot remove constraints to retest if that specific constraint was the cause without knowing the new ground truth. As potential change 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. In the other scenario when he 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

With deobfuscating the inputs, we can detect exact copies, but depending on the deobfuscation's time complexity other techniques could be better with similar results. In case where we would have access to stack traces, we could differentiate the bugs on the basis of the hash from the backtrace, sometimes even numerous hashes per input depending on the number of backtrace lines taken to hash. This technique is called "stack backtrace hashing" and is quite popular according to Valentin J.M. Manès et al. [39]. Another technique talked about in that paper, is looking at the code coverage generated by the inputs where the executed path (or hash of it) is used as a fingerprint of the inputs. A technique, used by Microsoft [17] is called "semantics based deduplication", where instead of backtrace they use memory dumps to hopefully find the origins of bugs. This use of dumps is less ideal due to traces having more information, but the latter is not always possible due to traces often being removed in production for performance and privacy reasons as specified in the paper. A last technique would be looking at the bug description left by manual user bug reports, although this dependence on the quality of bug reports and is most likely poorly automatable. None of the techniques mentioned above are perfect: with stack backtrace hashing you need access to the backtrace, with coverage some inputs will generate extra function calls and the semantics based deduplication are limited to X86 or x86-64 code with the binary file and the debug information. Neither of those first techniques will work with black box fuzzing unfortunately due to the limited information given as output.

4.3.2 The precision effect

The finding of the same bug needs to be done carefully, so that we do not change a null pointer dereference bug to a parser related bug. This, as discussed in the previous chapter, is because we value some bugs 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. But overall, we try to avoid this effect, which can be done with the techniques in the previous section. But 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 with it 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 more 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. With first starting off how we got our seeds to fuzz upon. Then, we will discuss how we implemented the three techniques to finally end with how we deobfuscated 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 simply because 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 to fast which resulted us moving to the academic license. For the other version of the solver, we used the once included in the already mentioned packages, except for MiniZinc's transformations to Google's OR-Tools[4]. This last one we had to install manually 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 of writing most examples are found in the main branch and some extras can be found

[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`

the "csplib" branch[5] waiting to be merged with the main branch. We downloaded a copy of that branch 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 [10] and got recommended by Ignace Bleukx. He 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 examples to test that the base examples do not crash on their own and noticed that most examples run in less than a minute. The handful of examples that did run extremely long we did leave out or simplified it to gain a speed up while solving of them. A last change we did to the future seed files is extracting the constraints from each example, 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. In order to extract these constraints, we temporary modified CPMpy to extract the created model, 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, which we altered to be able to find bugs in constraint programming languages and specifically CPMpy. We started with downloading 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. Unfortunate 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 introduces conversion bugs and the unclear way back from SMT-LIB prevented this path from being investigated by us.

Therefore, we decided to refactoring STORM to fit CPMpy and name the technique CTORM for CPMpy-STORM moving forward. To change STORM into CTORM, we need 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 incremental solving or the input obfuscation that was build-in. A bigger downside came with the refactoring of the negation function of STORM, as CPMpy is still in active development is not always available and that was felt while trying to negate global functions. 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

[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`

but vital to the fuzzer. So here we had the choice of adding the missing negation of global functions 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 cope 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. Therefore, we hijacked the flatten process of CPMpy to also return all subformulas before returning the flattened constraints. This gave access to the more convoluted subformulas to use in the next steps of CTORM before they are flattened. This flattening process was done before any modifications where 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 adds 100 new constraints built from the existing constraints. This is repeated a hundred times to create a hundred models to then check if the result match 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 changing them repeatably while keeping the (un)satisfiability the same. To then test if the original seed problem gives the same result as our modified problem. Matching with what we 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 functions like "alldifferent([var1, var2, var3])" to var1!=var2 and var1!=var3 and var2!=var3. Adding futile variables to global functions such as: "allequal([var1, var2])" either by copying variable or inserting new variable 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 [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 comparisons. And finally, 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 or other processes, but by combining multiple relations at random we were able to create more complex constraints that were not always handled easily. 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, more information can be found in the CPMpy bug report number 163.

5.5 Differential testing

With differential testing we stepped a bit further way from the fuzzing world since we did not edit the input files. With this last technique we put the (sub)solver against each other, if solver A said that the problem is unsatisfiable and all other solvers said that the same problem was satisfiable we could say that we did find a bug and that the bug was most likely to be in solver A. This 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. However, after finishing, we discovered that only two solver, namely "ortools" and "gurobi"-solvers, are able to search all solutions for problem models that contain global function. Small note: we did limit up to 100 solutions, otherwise finding solutions would take an unreasonable amount of time. More solvers are available to find all solutions for SAT problems but that is not the main objective of this thesis. This limitation to only two solvers results in only 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 able to automatically show us which of the solver is likely to be wrong. In the future more solvers will be available with the capability of finding all solution withing CPMpy, but right now these tests are performed with two solvers.

While searching for all possible solutions for a given problem over multiple solvers gave us more data to compare in between solvers the restrictions mentioned limit the testing of finding all solutions. However, when we look at only searching for one solution for a given example, we have more than enough solvers to compare between. Most of the times 14 solvers were compared and we never gotten lower than 6 solvers, this variation is due to the features used in the given problem the amount of solvers changed. If a problem happened to not use any global functions the SAT-solvers may be able to solve them allowing us to compare between up to 36 solvers. Given that we only search after 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 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 problem.

5.6 Detecting the cause of the bug

As describes in chapter 4 finding a bug is one part of the problem and submitting a complex bug would cause a lot of work for the development team. 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 advance 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 the

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

CPMpy's MUS-finder. We tried both 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. But 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 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[9] program suitable of finding MUS for all available solvers. Which was important since not all (sub)solvers would agree on the problem being satisfiable or not. And asking a solver to find an unsatisfiable subset in what it thinks is a satisfiable problem does not work 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 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 we used in tandem with the version's numbers and GitHub repositories where applicable for reproducibility. We showed how we turned Hakan's and CPMpy's examples into the seed files we 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 relations and the comparing of the output of (sub)solvers in the differential testing. To finally end with the tools used to minimize the input files once We found a bug.

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

Chapter 6

Results

In this chapter we will see the results of all three of the created techniques, we will explain how we prevented frequently occurring bugs, discuss a diverse subset of found bugs with more detail. And end the chapter with a classification of all bugs and the reception of the bugs.

6.1 Running the tests

Although, the specifications are not crucial since we did not do any speed benchmarking. It does give you, the reader, an idea of the performance. 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. 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

While the techniques were running, we noticed that once a bug was found by any of our techniques that same bug would occur frequently, but in a different seed causing our resulting logs to be cluttered with duplicates bugs. We were aware that this could occur and we originally wanted to use deduplication as described in subsection 4.3.1 to get rid of the equivalent bugs. But after seeing 10 000 logs of the same bug, we changed from a reactive approach to a preventative approach. Once we noted a bug 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 not log them. For example, a bug we will discuss soon had a specific string of characters, " == 0 == 0", making any problem wrongly unsatisfy. Being able to check the string of characters with knowing that it resulted in wrongly unsatisfiable solutions made it able 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, this is not an ideal solution but it worked well enough. The ideal solution would be that the fuzzer has knowledge of the already found bugs and reject them, techniques like these do exist but would bring us to far from the scope of the thesis. Although, less efficient the techniques could be used for detecting an error, fixing them and then rerunning for the next bug instead of adding exceptions.

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 bugs some of them were easy fixes, some were a bit harder and required more time to solve. Depending on which definition of a bug used the number of found bugs will differ with the definition followed in this paper 19 bugs remain out of the the 22 submitted. At the time of writing not al bugs are resolved (some are just reported days ago), but we look in anticipation how those will be solved. For now, let us look at some of the most interesting bugs we found. 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 Not

The first bug we discovered was our "double not"-bug, a bug where we ask CPMpy to solve the constraints "X==3 and not(not(X==3))". Clearly this solution is trivial, set variable 'X' equal to 3 and the problem would be satisfied. However not all CPMpy solver did agreed with this, both OR-Tools and Gurobi said that this problem was unsatisfiable.

This was due to a process within CPMpy responsible for creating a flat normal form. Not all solver used by CPMpy allow an arbitrary nesting of constraints as described by the documentation of CPMpy[1]. It is for that reason that CPMpy flattens the constraints to what they call 'flat normal forms' as the similar definition of SAT. But, with a disclaimer that this definition does not formally exists for CP languages to their knowledge, a statement which we agree with. With this flattened form CPMpy can 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].

Those not's get translated to two comparison with a zero "== 0", but in the normalizing process that a comparison withing a comparison was not handled correctly. Causing a disappearance of a single "not", which in turn resulted the original constraint converted to "X==3 and not(X==3)". When this gets sent to OR-Tools or Gurobi they correctly say that the problem is unsatisfiable. The other solvers, mainly MiniZinc's subsolvers, were not affected by this bug due to not using this normalizing process. Although, this normalizing process was subjected to unit

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

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 not's and and's. But we believe it could have been caught in the metamorphic testing if we had thought of adding a relevant metamorphic transformation or in the differential testing if a seed had a double "not" in its constraints.

A showcase of this double not bug can be seen in listing 6.2.1, 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.

```python
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 not"-bug.

6.2.2 Negation of Global functions

A second bug related to the use of not's were the crashes of the negated global functions. Where negations of global functions like "not(AllDifferent(argList))" would crash with a maximal recursion depth. As the normalizing of a negated global function would be handled with adding a "== 0" to it, instead of decomposing the global function and negating that. The action of adding a "== 0" did not change the constraint and on the next normalizing of the left part of the comparison "global function == 0", the same would happen. The solution was, as mentioned to decompose the global function, which was suggested in the comments together with a commented out throw of an 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 functions 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. And this bug was quickly found by CTORM since it uses a significant number of not's. Due to the metamorphic relation of adding "!= 0" after some constraints the metamorphic tests did find it as well. However, it did not get found by our differential tests simply because no examples negated their global functions, as such a negation is rarely useful.

A showcase of this bug can be seen in listing 6.2.2, 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 function "AllDifferent" is added to a created model on line 5. With this we ask to find not all different values of the array "pos", which will be satisfies as long as one of the elements withing the array has the same value as one of the other elements in the array. For example, array "[1 2 2]" would satisfy. Subsequently, a MiniZinc subsolver is used to solve the problem which returns feasible on respectively line 7 and 8. However, when asking the same for solver Gurobi it will crash, analog 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 6.2: The "negation of global functions"-bug.

6.2.3 Power function of Gurobi

Now that we have seen two bugs in the normalization 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. A 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. A lower bound meaning that the variable was not permitted lower that that bound. All other solver 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. But Gurobi did (and still does) not allow this throwing an error as a result as can be seen in listing 6.2.3.

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 written. The original example can be found at the CPMpy repository's csplib examples[2]. We do not think that CTORM would have found it, if it was not in the seed file to begin with. This because CTORM does not create new

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

variables nor modifies any bounds of variables. The bug was also not found using the metamorphic tests, but we do believe that if we had written a metamorphic relation for the power function or one where we changed some bounds of variables that we could have found this bug. And 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.

```
from cpmpy import *

m = Model()
X = intvar(lb=-5, ub=5)
m += pow(X, 2) == 9

m.solve(solver="gurobi") # GurobiError
```

Listing 6.3: 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 asking for a check if a sum of booleans matches a specific variable and that variable happens to be an integer instead of a boolean naturally the SAT solver, which only support boolean satisfiability problems, will complain. 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, but got a larger bounds. On all other places we could find check with an error that would be thrown. But 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. This check would often crash with PySAT's subsolvers since almost all seeds were written with a CP solver in mind. 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 this 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 that 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 2.2.3) 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 2.2.3 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 because reusing variable names is dangerous when 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 got 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.

A similar and almost related bug is in the naming of variables, when creating them starting with strange symbols like ’+’, ’%’ or others some solvers would crash. Most solvers would happily solve with these names, but all MiniZinc’s subsolvers crashed with a syntax error when handling the input. This due to transformation of our model to the text-based FlatZinc 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 6.2.5 this bug of strange 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 strange naming of variables MiniZinc will crash with a syntax error on line 10 while it solved the constraint fine on line 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 m.solve(solver="minizinc:chuffed") # crash by syntax error
```

Listing 6.4: A bug showcasing that the naming of CPMpy is looser then MiniZinc.

Both of these bugs were not caught by CTORM nor the differential testing, since they do not create new variables. But did get caught by the metamorphic tests, 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 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. This 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 subsolvers Gecode and others it 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.

optional: bug 163 would be a fun explanation for the thesis, but lacks a bug. atm it's a note in 5.4

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 withing CPMpy. With 4 bugs in the model, 7 bugs in the general transformations, 7 bugs in the solver interface and one a solver related bug. The one and only solver related bug we found was the one we discussed in subsection 6.2.6, which was already known by MiniZinc. We would have hoped to find more bugs in the solvers themselves and it was our aim with this thesis. But 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 withing 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 have used the techniques used in this thesis. This 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

Table 6.1: Table discussing in which CPMpy component the bug was found. With 4 bugs in the model, 7 bugs in the transformations, 7 bugs in the solver interface and one a solver were found.

BugNr	Bug description	Place of the bug
142	double not gives unsat	Transformations
143	negating global functions 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

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. Were we see 5 bugs unrelated to any solver, that OR-tools only occurred together with Gurobi and that OR-Tools and Gurobi didn't share any bugs found with MiniZinc or PySAT. Meanly because OR-Tools and Gurobi share more transformation code than any other solver. We also see that Gurobi occurs the most among out our bugs, this often due to edge cases on Gurobi's implementation. Although, it could be coincidental that we found more Gurobi bug than any other we speculate that the software being proprietary makes it less clear-cut to be implemented in CPMpy.

The last table 6.4 shows the techniques finding which bug. CTORM found 10 bugs, metamorphic testing found the most bugs at 13 and differential testing found 11 out of the19 found bugs. This shows that none of the techniques are perfect on their own, in order to find all bugs a combination of techniques would be needed. As the industry's quality insurance processes do for finding bugs in other software packages, where they use a combination of tools as discussed in section 1.1. With

TABLE 6.2: Table discussing what type of fault was caused by which the bugs.

BugNr	Bug description	Type of fault
142	double not gives unsat	wrongly unsat
143	negating global functions 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

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 thrown 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 could perhaps have seen some negative opinions on fuzzing. Although, we submitted 22 bugs or questions withing 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 not"-bug[3] was called a serious bug and a great find and the "negation of global functions"-bug[4] was described as an unexpected bug and another great find.

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

Table 6.3: Table discussing which bug was caused by which solver or if it was a solver independent bug.

BugNr	Bug description	Which solver caused it?
142	double not gives unsat	OR-Tools and Gurobi
143	negating global functions 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

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. But after filtering previously found bugs the techniques perform 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. We have seen that most bugs were related to the CPMpy code and only one responsible by an external solver. We suspect this lack of external solver bugs to be caused by well written and tested solvers. We have shown that we found 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. And finally, we noted the grateful welcome of bugs by the CPMpy-team.

TABLE 6.4: Table discussing which technique found the bug. 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 not gives unsat	ctorm		
143	negating global functions 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	

Chapter 7

Conclusion and future work

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

proofread chapter

7.1 Achievements

This thesis achieved to find numerous bugs withing CPMpy this with the use of multiple techniques. As discussed in section 6.3, some bugs where 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 where found in the solver themselves. A third of the time the bugs resulted in a critical bug, this being a 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 case 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 need to be made as clarified in subsection 6.3.

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. But 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 not"-bug and the power bug of Gurobi as well. But 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 way from automation.

While working with seeds is the better than generating inputs the 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 functions was not found using differential testing since none of the over nine thousand seed files had any negation of a global function. 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 preformed study. This testing of configuration space was shortly 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 options of the PUT. Within CPMpy a lot of extra solver parameters are available depending on the solver used, which form a nice basis to fuzz the configuration space on top of the inputs. 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 - Courese Text*. 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 concistent editing result indepentent 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.