Digital Forensics Research Using Constraint Programming - A preliminary approach

João Calhau, Pedro Salgueiro, Salvador Abreu, and Nuno Goes

Universidade de Évora

Abstract. The world today is becoming more and more digital, as such there is a huge amount of data constantly being created, transmitted and saved every second. Representing this data in digital form, although it brings some advantages it also brings some disadvantages and challenges when we need to make an analysis of the content of said data. For this same reason, it was created the discipline of digital forensics, which focuses on the analysis of digital equipment content. This document introduces an approach on using constraint programming methods in digital forensics analysis, allowing for an easier and more efficient method to analyze digital equipment data.

Keywords: Digital Forensics, Contraint Programming, Security, Declarative Programming

1 Introduction and Motivation

In this work we present a system that makes use of the constraint programming paradigm and methods to solve digital forensics problems, developed in the context of João Calhau MS Thesis. Through the use of constraint programming, we're able to describe a digital forensics problem in a declarative and expressive way, and reach a solution to the problem. The digital forensics problems we're trying to solve are problems such as discovering files that have a specific name, path, type or content and timelines.

The main purpose of the system presented in paper is to allow for an easy and efficient method to search for relevant information in the contents of digital equipment, as previously described. Beyond that, the system must also be able to detect important dates, snapshots, restore points or versions, all these operations mentioned must be performed in a chronological order and without any previous pre-processing of the digital image.

1.1 Constraint Programming

Constraint programming is a powerful paradigm mostly used to solve combinatorial problems. It looks like a simple way to model real world problems but can actually turn into a complex challenge when we want to find solutions for the

problem that is being solved. Constraints can be found in our day to day experiences, almost ubiquitous, representing the conditions that restrict our freedom of decision [15].

In constraint programming, we usually have a set of variables, with an initial domain, to which constraints are applied in order to reduce its domain, and thus reach a solution. Once a constraint is placed on the system it cannot violate another constraint previously applied. This way we can express the requirements of the possible values of the variables [13].

Constraint satisfaction problems are usually solved with the help of solvers. These solvers are essentially search algorithms, usually based on backtracking techniques[11], constraint propagation [12] or local search [6].

1.1.1 Backtracking

Backtracking is a search method that incrementally finds possible candidates to solve the problem. At the same time it removes the candidates that can not be used as a valid solution to the problem [11]. One of the most used examples for this type of search method is the n-queens puzzle, where a set of n queens should be organized, in a $n \times n$ chess board, in such a way that none of the queens can attack each other. Any partial solution that contains two queens that can attack each other is abandoned immediately.

1.1.2 Constraint Propagation

Constraint propagation starts by reducing the variable's domain, strengthening or creating new constraints, reducing the search space, which leads to a problem that is easier to solve. Since this algorithm only reduces the search space reduction of the problem variables, after completion, there is still the need to use another algorithm to solve the problem, which was converted into a simpler problem by the propagators [12].

1.1.3 Local Search

Local search is an incomplete search method to find solutions for a problem. It consists in, iteratively, and with the help of previously defined heuristics, assigning values to the system variables until all the constraints are satisfied. At each step of the iteration, the values of the variables are updated to values *near* the previous value. The algorithm also makes sure of the quantity of constraints it violates so it can have a "cost" associated with the attribution of the values and it can thus tell us if a pre-determined cost has been met [6].

1.2 Digital Forensics

Digital forensics is a very important discipline in criminal investigations where the main device used was an digital device, or to investigates crimes where the evidence may be stored in a digital device. The digital forensics tools have

become a vital appliance to assure we can rebuild information after a cybernetic attack or even if we just want to analyze any type of digital equipment. [8]

It's a complex task to collect evidence/elements in digital equipment, either connected to a criminal activity or not. If that piece of equipment is connected to any type of computer network, with the consequent increase in digital traffic, more difficult it becomes to detect any anomaly or undesirable communication in the network. Thus, the intrusion detection systems have become a very important tool in computer network security. [16]

To collect evidence/data in digital equipment there are many tools capable of analyzing a digital forensics image. Among them, one of the tools that is most used is the *EnCase Forensic* [18], also used in several judicial systems. Besides this tool, which is proprietary and commercial, there are other open source and free of access tools capable of accomplishing the same work. The *Forensic ToolKit (FTK)* [1] and the *Autopsy* [2] are two examples.

2 Known Tools and Other Approaches

In this section we introduce the practical aspects of constraint programming including some libraries and toolkits that are used to model and solve constraint problems. We also describe similar work already done in this area.

2.1 Choco

Choco is a free access and open source library dedicated to constraint programming. It is written in Java and supports several types of variables, including Integers, Booleans, Sets and Reals. It also supports several types of constraints such as AllDifferent and Count, configurable search algorithms and conflict explaining. The first version of Choco was developed in the early 2000s. A few years later, Choco 2 was developed and declared a success in the academic and industrial world. Since then, Choco has been completely re-written and in 2012 the third version of Choco was launched. The current version comes with a simpler API and is denominated Choco 4 [14].

2.2 Gecode

Gecode is a free access, open, portable, accessible and efficient programming environment used to develop systems and applications based on restrictions. Gecode, much like Choco, supports various types of variables and restrictions, among them are Integers, Float and Sets. These variables are used to model problems that are then solved with the help of constraint propagators and search algorithms [17] [9].

2.3 Google OR-Tools

Although Choco and Gecode are two of the most widely used libraries, there are also other new tools, such as the Google OR-Tools. Google Optimization Tools

or OR-Tools is an interface that puts together several linear programming solver and that counts on the use of several types of algorithms such as search algorithms and graph algorithms. What this library has that is so noteworthy is the fact that it doesn't let itself be bound by one language. Although implemented in C++, it is capable of working in other languages like Python, C# or Java.

2.4 The Sleuth Kit

The Sleuth Kit is C library and a collection of tools that allows its users to analyze disc images and restore files from it. The Sleuth Kit is what Autopsy [2], the forensics tool mentioned earlier, uses in it's background jobs. The Sleuth Kit framework allows the user to incorporate additional modules so he can analyze file contents and build automated systems. In addition, the library can be embedded in larger digital forensics tools and command line tools can be used directly to find any kind of proof [3].

Of all the tools The Sleuth Kit has to offer, the most interesting to help us in the type of problem we're trying to solve is the Sorter, which analyzes a file system and organizes what it finds by extension of file. In addition, it provides us details about the organized files, such as the file inode number. The Sorter can also use a separate hash database to ignore files that are known to be good, such as Dynamic-Link Libraries, or dlls, of the windows file system or even know applications.

2.5 Digital Forensics and Constraint Programming

During the analysis and study of the state of the art about constraint programming and digital forensics, no studies were found that combined the two areas. The topic that most resembled this was the use of constraint programming in artificial intelligence in order to make cyber-defense a more reliable and secure method. [19] Considering the reduced number of related works, it is part of the proposed work to see if the use of programming by constraints introduces improvements in terms of processing speed and ease of finding clues or evidence in the cyberspace.

3 Approach

This section introduces our approach for modeling a Digital Forensics Problem as a Constraint Satisfaction Problem using the Choco Solver. We include all data structures needed to model the problem, how they are used to reach a solution to the problem, the methodologies used to analyze and extract the information from the digital evidences, and how the problem is modelled as a CSP.

3.1 Methodology

After acquiring the disk image to be analyzed, it is first processed by the Sorter tool from The Sleuth Kit [3]. The Sorter outputs multiple files with different

names, each name being a pre-determined type of file, such as archive, executable or data. Each output file contains all the information the Sorter collects such as the file path in the file system, the file type, the image name (from where the data was extracted) and the inode number, which is an internal representation of that particular file in the file system. With this information there isn't much we can do, because the information hasn't been processed yet, this means we need to store it in some kind of data structure. When thinking about what type of data structures to use we figured out that having the data organized in various different ways would make it easier to extract the information in a later date, so, instead of one data structure we decided on using three different data structures. These data structures are described in Section 3.1.1

To populate these data structures we parse the data of files created by the Sorter tool. That can be easily achieved by reading the contents of said files and storing them in some kind of "wrapper" in the data structures. This wrapper ended up being a representation of the inodes containing the inode number, the file path, the file type and the file name. The files outputted from the sorter always have the same type of structure, three lines of text followed by and empty line, this can be seen as an example in figure 2. Also the three lines of text always come in the same format, first line is file path and name, second line is file format and third line is image name and inode number. All we have to do is iterate over those lines four at a time, gather the information, build the inode representation and store it on the data structures.

The whole process of extracting the data from the file system can be shortened into the small flow diagram seen in figure 1.

Fig. 1. Flow diagram

```
1  idle_master/CSteamworks.dll
2    PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
3    Image: pen_4_dd.dd  Inode: 40-128-1
4
5  idle_master/HtmlAgilityPack.dll
6    PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows
7    Image: pen_4_dd.dd  Inode: 42-128-1
8
9  idle_master/IdleMaster.exe
10   PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
11   Image: pen_4_dd.dd  Inode: 43-128-1
```

Fig. 2. Example of sorter output for executable files

3.1.1 Data Structures

As said before, we decided to go for three different data structures, the first one would be used to store the representation of the inodes, these didn't need to be stored in any particular order so we went for a data structure that didn't have order, that was easy and efficient to use, an Hash Map of inodes. In the second structure we decided to store the inode representations, but this time by path, that is, we use the same type of structure as the first one, but with a twist, this time we would use a Hash Map of Linked Lists of inodes because they are going to be organized without any particular order, but more than one inode can have the same path, this means we would need to insert various inodes with the same key in the Hash Map, and that is just not possible, unless we insert the inode in the linked list at that key position. Finally, the third data structure would be used to store the inodes by type and as the file types are always the same (because the sorter always outputs the same type of file types) we decided on using a simple array (of fixed position) of Linked Lists, the array is always the same, what changes is the Linked Lists inside said array.

3.2 Modelling

As previously mentioned, the files extracted from the file system are sorted into various different types and have various types of data extracted from them, such as type, path, name and inode number. Because each file has a different inode number, which is an integer, we decided to use the inodes to represent our files in our Constraint Satisfaction Problem.

The framework we ended up deciding on using was Choco, due to it's good rating, good documentation and language familiarity (Java, when opposed to C++). We now had our variable's domain, all the inode number in the file system, but we still did not know which type of variable we would use. Choco has four different types of variables available with which we could model our problem with: Integers (IntVar), Booleans (BoolVar), Sets (SetVar) and Reals (RealVar). We wound up deciding on Integer Set Variables, or SetVars, because the final solution of our solver would need to be a set of one or more integers.

In Choco Solver, SetVars are defined by a domain that is composed of two separate domains, the LB and the UB, these being Lower Bound and Upper Bound, respectively. The Lower Bound is a set of integers that must belong to every solution and the Upper Bound is composed of the set of integers that may be part of the final solution [5]. In our case, when creating the variable with which we are going to work with, the Lower Bound will be left empty, because we do not know what files we want yet, and the Upper Bound will be composed of all the inodes existing in the first data structure (the Inode Data Structure). Finally all that is left to do is create our custom constraints and apply them to our variable so we can restrict it's domain.

3.3 Constraints and Propagators

To implement a new constraint in Choco solver, first we need to create a propagator. A propagator declares a filtering algorithm that can be applied to the Variables that model the problem, in order to reduce their domain [4]. Since this work is still in it's early stages we decided to create a couple of simple propagators to test the validity of our approach. We decided to implement the following propagators: 1) file type propagator that restricts our domain according to the given type of file passed as argument, it restricts the domain based on the type data structure created before; 2) file path propagator that restricts our domain according to the path passed as argument, it restricts the domain based on the path data structure created before. Both propagators were build to work with the variables used to model the problem, SetVars.

For the propagators to work, we have to implement the following methods methods: `propagate` and `isEntailed`. The method `propagate` is pretty straight forward, it should restrict the domain according to what we need. The method `isEntailed` is straight forward as well, all we need to do here is tell the propagator when the problem has a solution or not, or if it is simply undetermined. These methods are described in detail in Listings 1 and 2.

Listing 1 propagate method

```
for each value in UB do
    if value is not in corresponding structure then
        Remove value from UB
    end if
end for
```

Listing 2 isEntailed method

```
if UB is empty then
    Problem is impossible to solve
else
    Problem has possible solution
end if
```

Both propagators work in a similar way, they take the Upper Bound of the SetVar, iterate over it and remove any inode that is not in the desirable data structure. For example, if we want to propagate an executable type, our SetVar is composed of two domains: and empty one {} (the Lower Bound) and one with three inodes $\{100, 101, 102\}$ (the Upper Bound). Our type structure only has one inode in the executable division $\{100\}$, what the propagator would do in this situation is remove every inode that is not in the type structure from the SetVar which would leave it like so: $SetVar = \{\}, \{100\}$. Of course this is

only an example, and on top of this another constraint could be applied, like a path constraint (never another type constraint, because that would just leave the Upper Bound domain empty every time).

3.4 Experimental results

In our experimental phase we decided to start by analyzing something small, like a disc or a pen drive, what ended up being chosen was the latter. We took a simple four gigabyte pen drive and passed it through FTK Imager [1] to obtain it's image. After obtaining the image we wanted, we passed it through the Sorter and waited for the contents of the pen drive to be sorted. After the Sorter finished it's work we would have to choose the constraints we would want to use later on.

For a first test we chose only one constraint, a type constraint that would restrict the domain to only the files that had an archive type. We booted the program and, sure enough, the resulting solution outputted only the inodes belonging to the files that had an archive type as can be seen in figure 3. Seeing as the first test went well, we decided to apply a second constraint on top of the first one, this time we chose the unknown type for the type constraint and a specific path in the file system "LVOC/LVOC/", this way the program should output only seven file inodes and it did as observed in figure 4.

```
D:\Java\jdk-8\bin\java ...
Inodes found:
Inode(45, idle_master.zip, idle_master, Archive)
Inode(49, LVOC.zip, LVOC, Archive)
Inode(717, Exclusive Collection of E3 2016 Cards.zip, ubi 30, Archive)
Inode(718, Exclusive Digital Posters from E3 2016.zip, ubi 30, Archive)
Inode(719, For Honor GIFs.zip, ubi 30, Archive)
Inode(720, Ghost Recon Wildlands GIFs.zip, ubi 30, Archive)
Inode(721, Holiday Wallpaper.zip, ubi 30, Archive)
Inode(722, Just Dance Greeting Card.zip, ubi 30, Archive)
Inode(723, Rabbids Holiday Goodies.zip, ubi 30, Archive)
Inode(724, Rayman GIF.zip, ubi 30, Archive)
Inode(725, Ubi30 360 Image.zip, ubi 30, Archive)
Inode(726, Ubi30 Exclusive GIF.zip, ubi 30, Archive)
Inode(727, Ubisoft Cocktail Recipes.zip, ubi 30, Archive)
Inode(728, Ubisoft Dessert Recipes.zip, ubi 30, Archive)
Inode(729, Ubisoft DIY Advent Calendar.zip, ubi 30, Archive)
Inode(730, Ubisoft Gift Tags.zip, ubi 30, Archive)
Inode(731, Ubisoft Wrapping Paper.zip, ubi 30, Archive)
Inode(732, Wallpaper for Mobile.zip, ubi 30, Archive)
Inode(733, Watch_Dogs 2 Wallpaper.zip, ubi 30, Archive)
Inode(734, Werewolves Within Wallpaper.zip, ubi 30, Archive)
```

Fig. 3. Program output for the first test

```
D:\Java\jdk-8\bin\java ...
Inodes found:
Inode(40, CSteamworks.dll, idle_master, Exec)
Inode(42, HtmlAgilityPack.dll, idle_master, Exec)
Inode(43, IdleMaster.exe, idle_master, Exec)
Inode(46, Newtonsoft.Json.dll, idle_master, Exec)
Inode(47, steam-idle.exe, idle_master, Exec)
Inode(50, Steamworks.NET.dll, idle_master, Exec)
Inode(51, steam_api.dll, idle_master, Exec)
```

Fig. 4. Program output for the second test

4 Conclusion and Future Work

From the experimental phase, we can conclude that what took the most time to finish was the extraction of the image with the help of FTK Imager [1], this took about five minutes for a four gigabyte pen drive that had about two gigabytes of data. The Sorter ran in about twenty-five seconds while the Java program ran in under one second. We can also conclude that the program works and restricts the domain as it should, in a very short amount of time, with both devised constraints placed at the same time. What we would need to do in terms of future work, would be the creation of more constraints, like restricting the domain further to files that only have certain keywords or files that have been altered recently or in a certain period of time.

References

1. AccessData. Forensic toolkit, 2017.
2. Brian Carrier. Autopsy - the sleuth kit, 2017.
3. Brian Carrier. The sleuth kit, 2017.
4. Choco-Solver. Class propagator api, 2018.
5. Choco-Solver. Interface setvar api, 2018.
6. Rina Dechter. *Constraint Processing*. 2003.
7. Edward Delp, Nasir Memon, and Min Wu. Digital forensics [From the Guest Editors]. *IEEE Signal Processing Magazine*, 26(2):14–15, 2009.
8. Simson L. Garfinkel. Digital forensics research: The next 10 years. *Digital Investigation*, 7(SUPPL.), 2010.
9. Gecode Team. Gecode: Generic constraint development environment, 2006.
10. Google. Google optimization tools, 2017.
11. Donald E. Knuth. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 1997.
12. Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms*. 2010.
13. Justin Pearson and Peter Jeavons. A Survey of Tractable Constraint Satisfaction Problems. pages 1–42, 1997.

14. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.
15. F. Rossi, P. Van Beek, and T. Walsh. Handbook of Constraint Programming (Foundations of Artificial Intelligence). pages 281–322, 2006.
16. Pedro Salgueiro, Daniel Diaz, Isabel Brito, and Salvador Abreu. Using constraints for intrusion detection: The NeMODe system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6539 LNCS:115–129, 2011.
17. Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling. In Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2017. Corresponds to Gecode 5.1.0.
18. Guidance Software. Encase forensic, 2017.
19. Enn Tyugu. Artificial intelligence in cyber defense. *2011 3rd International Conference on Cyber Conflict*, pages 1–11, 2011.