UCLouvain

epl

École polytechnique de Louvain

# Graphical user interface for DAG-based attack trees

TFE22-024 Handling attack in an efficient way!

Authors: **Florentin Delcourt, Thomas Weiser**
Supervisor: **Axel Legay**
Readers: **Charles Pecheur, Pierre Martou, Sébastien Jodogne**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

# Abstract

The thesis described in this document has been divided into two different parts. First, a deep research on the topic of attack trees to understand the theoretical background and the different ways to compute and analyse such structures. The second part of the thesis covers the creation of a tool to compute and analyse attack trees.

The attack tree model owes its popularity thanks to two key ideas : its readability and its ability to cover a large domain of applications. The simple principle behind attack trees is to break down complex scenarios or systems into smaller interconnected scenarios that can be assessed and quantified more easily. On top of that, they are convenient to perform analysis to compute and classify the different attack traces of the system and determine the more efficient countermeasures.

In the 1990s, different groups were developing modeling concepts called *threat trees* and *threat logic trees*. The term attack trees has been popularized in 1999 by a cryptographer and security expert named Bruce Schneider [72]. Over the years, the model evolved through different researches proposing many formal semantics for attack trees [37][42][58]. The attack tree model has also been expanded and mixed with defense tree models to create attack-defense tree with the addition of countermeasures [47][71]. A wide overview of attack trees has been written by Kordy et al. [48] in a Computer Science Review.

System security against attacks is a major concern and sees a wide variety of application areas. Some documents relate the use of attack tree graphs to analyse the security of critical infrastructures in the electric sector [62], to classify ATM-related frauds [31] or even assess vulnerabilities in SCADA systems [20]. The recent trend uses the attack tree design models and enhances them with analysis methods like Bayesian networks [34], model checking [14], automata theory [51], SAT and SMT solver for constraint solving [15], ...

This thesis uses a DAG-based attack tree model to assess attack risk and uses previously implemented SAT and SMT solvers to compute tree solutions with constraint programming. May this document be an additional use case of this model.

# Contents

# List of Figures

# Abbreviations

**ADTrees** Attack-Defense Trees

**CDCL** Conflict-Driven Clause Learning

**CNF** Conjunctive Normal Form

**CNF-SAT** Boolean satisfiability problem restricted to CNF inputs

**CWMP** CPE WAN Management Protocol

**DAG** Directed Acyclic Graph

**DDoS** Distributed Denial-of-Service

**DFS** Depth First Search

**DLL** Dynamic Link Library

**DNF** Disjunctive Normal Form

**DP** Davis–Putnam algorithm

**DPLL** Davis–Putnam–Logemann–Loveland algorithm

**EUF** Equality and Uninterpreted Functions

**FTP** File Transfer Protocol

**GrSM** Graphical Security Model

**GUI** Graphical User Interface

**HTTPS** HyperText Transfer Protocol Secure

**IFF** If and only if

**IoT** Internet of Things

**K-Map** Karnaugh mapping

**LIA** Linear Integer Arithmetic

**NNF** Negation Normal Form

**NP** Non-deterministic Polynomial time

**NRA** Non-linear Real Arithmetic

**P** Deterministic Polynomial time

**QF** Quantifier Free

**QMC** Quine–McCluskey

**SAT** Boolean satisfiability problem

**SAT solver** Tool checking boolean SATisfiability

**SLS** Stochastic Local Search

**SMT** Satisfiability Modulo Theories

**SSH** Secure Shell

**SYN packet** SYNchronize packet

**TCP** Transmission Control Protocol

# Part I

# State of the art

# Chapter 1

# Attack tree

## 1.1 Introduction

This chapter will present existing published work in the field of risk assessment using graphical models. To ensure the security of complex systems and devices, many different models have been proposed over time in order to unify two key ideas : (1) return a simple and effective representation of possible complex attack scenarios and (2) allow this structure to perform qualitative and/or quantitative analysis so that the defender can draw conclusions and mitigate incoming threats. The model describing attack trees is closely related to the two other models we will refer to in this thesis. The first one concerns feature models used to describe relations between mandatory and optional features in a system diagram. The second one is an extension of the attack tree model called attack-defense tree involving the behaviour and actions of the defender. This adds countermeasures and defense nodes to the model to express the actions of the defender of the system.

## 1.2 Feature model

Feature diagrams are a popular and valuable tool to engineer software product lines. Many researches tried to formalize the syntaxes of feature diagrams to avoid ambiguities. Schobbens et al. [74] wrote a survey in 2006 to propose a generic formalization of the syntax and semantics of feature diagrams by formally comparing languages and studying decision procedures. The second main resource we will use was published by Krzysztof Czarnecki and Andrzej Wasowski [22] to establish the link between feature models and propositional formulas enabling analysis with logic-based tools. They mainly worked on the translation problem considering the extraction of feature models from propositional formulas.

### 1.2.1 Formal definition

Using the definition of basic feature model written by Krzysztof Czarnecki and Andrzej Wasowski [22], a feature model is a tree of features representing a system. The root node of the tree represents the system to be designed. The remaining nodes are grouped features or solitary features. Grouped features can be contained by xor-groups or by or-groups while solitary features are stated as mandatory or optional. At last, additional propositional formulas can be used to specify supplementary constraints on the model.

Here is a small example from the same document. The car is the root feature representing the main feature of the production line. There are some grouped and solitary features and one additional propositional formula stating that the presence of a *keyless_entry* implies *power_locks*.

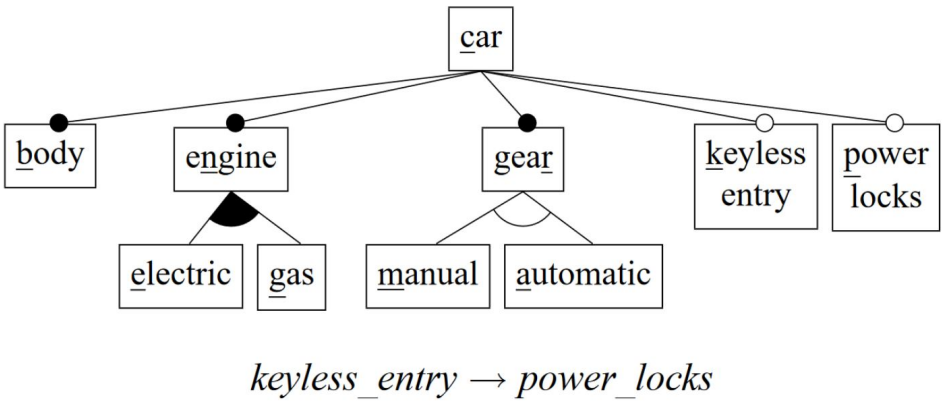



Figure 1.1: Basic feature model (reproduction from [22])

The following table 1.2 described the modeling rules of this type of feature model. It represents the list of symbols needed to build and understand the graph. This example illustrates the expressiveness of such tools despite a relatively small syntax.

| Symbol | Explanation |
|---|---|
|  | Solitary feature with cardinality [1..1], i.e., *mandatory* feature |
|  | Solitary feature with cardinality [0..1], i.e., *optional* feature |
|  | Grouped feature |
|  | Feature group with cardinality $\langle 1\text{–}1 \rangle$, i.e. *xor*-group |
|  | Feature group with cardinality $\langle 1\text{–}k \rangle$, where $k$ is the group size, i.e. *or*-group |

Figure 1.2: Feature model syntax (reproduction from [22])

### 1.2.2 Propositional logic

Any feature model can be translated to a boolean formula whose solutions will define all possible configurations (a set of selected features) creating the root feature according to the constraints. This mathematical representation is the basis used to perform qualitative and quantitative analysis on the feature model and assess them. There is the propositional formula representing the car example.

$$
\begin{aligned}
& q_{FM} = \\
\textit{child-parent:}\quad & (b \rightarrow c) \wedge \\
& (n \rightarrow c) \wedge (e \rightarrow n) \wedge (g \rightarrow n) \wedge \\
& (r \rightarrow c) \wedge (m \rightarrow r) \wedge (a \rightarrow r) \wedge \\
& (k \rightarrow c) \wedge \\
& (p \rightarrow c) \wedge \\
\textit{mandatory:}\quad & (c \rightarrow b) \wedge (c \rightarrow n) \wedge (c \rightarrow r) \wedge \\
\textit{or-group:}\quad & (n \rightarrow e \vee g) \wedge \\
\textit{xor-group:}\quad & (r \rightarrow m \veebar a) \wedge \\
\textit{additional:}\quad & (k \rightarrow p)
\end{aligned}
$$

Figure 1.3: Feature model formula (reproduction from [22])

This short presentation of the feature model subject can be deepened with the following documents : Schobbens et al. [74] defined a generic semantics of feature diagrams and Krzysztof Czarnecki and Andrzej Wasowski [22] published a way to recover feature models from a propositional formula. This basis evolved toward the attack tree model described in the next section.

## 1.3 Attack tree

Attack trees are based on the feature models just described, but instead of describing an assembly line, they expose the situation of an attacker seeking to harm a system. The key idea behind attack trees is to provide a simpler visual representation of a possible complex attack against a system by decomposing it into multiple intermediate actions until they become a composition of atomic basic actions. In addition to attack path computation, this structure allows quantitative assessment as these atomic actions can be more easily evaluated. The goal of risk assessment is to provide the best countermeasures to the defender against the actions of the attacker. As general objects, they are useful in various situations in the industry as described in the abstract of the document. A large literature explores the ways to use and design tools expressing attack trees. A global review of the topic published by HS Lallie et al. [52] in 2020 is a great document covering this domain.

### 1.3.1 What is a cyber attack

Several conditions must be met for an attacker to successfully perform an attack. The victim's system must suffer from certain vulnerabilities or weaknesses, the threat agent must have enough resources to use these weaknesses and bypass the security walls and finally the attacker must gain advantages if the attack is successful. A document [39] published in 2010 cites different motivations an attacker can have while harming a system and state that an attack risk is worth depending on two factors : the success probability and the impact on the victim. This section goes through different high-level components of the cyber attack literature.

#### Attack components

The Internet Security Glossary written by R. Shirey [75] provides definitions of words and concepts from the security vocabulary. Some necessary terms are defined in this section to clarify the further discussion about attack trees.

**Vulnerability :** It designs a weakness in a system's design, implementation or management that could be exploited to violate the system's security policy [75]. A system usually suffers from multiple vulnerabilities but any vulnerability does not necessarily lead to an effective and harmful attack. This depends on the scale of the vulnerability and the degree of the attack. Complex and innocuous vulnerabilities are not critical and can be tolerated, while easy-to-execute attacks should be avoided.

**Preconditions :** Prerequisites for an exploit are the necessary conditions that must be met for the exploit to be feasible [9]. Initial preconditions are present due to the system and not due to previous exploits. These often come from the system itself like the actual versions of the OS, some applications or some services. The prerequisites also take into account the reachability of the target and the ability of the attacker to execute a process on this target.

**Postconditions :** The postconditions are the results or the goals of a successful cyber attack. These could be used as preconditions for other exploits in the case of multi-stage attacks.

**Exploit :** The exploit is the sequence of actions perpetrated by the attacker during a cyber attack. Under the preconditions of the attack, the exploit abuses of the vulnerabilities of the system in order to successfully perform the attack and access its postconditions.

**Attack tree :** An attack tree represents an attack scenario against a system with different components. The root node of the tree is the goal of the attacker and can be achieved by combining different actions on the components of the system. Hence, multiple exploits can be expressed by a single attack tree regarding the complexity of the system and the precision with which the tree was built.

This non-exhaustive list of components treated some basic terms present in the cyber security landscape. The next section defines and places the model of attack tree inside this cyber security landscape.

### 1.3.2 The representation and meaning of attack trees

In practice, attack trees were constructed manually by security experts, leading to several serious drawbacks [80]. Indeed, manual construction depends on the skill and expertise of the creator. Two different tree builders would probably propose two very different attack trees to describe the same system. They might not consider relevant the same actions or components depending on the attack tree model they are using. These trees describing a unique system might differ in size and structure providing different analysis results. Many survey documents enumerate and classify the different models proposed by the community trying to unify the notations and features. The good point about attack tree is that roughly all models stand on a shared basis that this section is about to describe. The definitions, modeling and features presented in this section result from our knowledge on the topic but also depend on our needs as this thesis aims at the creation of a tool to model and analyse attack trees. Many other aspects could have been discussed and can be seen in the related work section 1.6.

**Attack tree definition :** An attack tree is a rooted tree structure with labeled nodes. These labels represent steps or intermediate goals for the attacker to achieve the main goal node labeled on the root of the tree. Basic attack trees can be represented as a tree node structure, but further sections use an upgraded *directed acyclic graph* (DAG) visual structure to model more complex features. Two types of nodes can be found in basic trees :

**Leaf node** : A leaf node is a basic node labeled with a basic atomic action and can not be decomposed in any further levels of nodes. In an attack trace, a leaf node can either be used (true) or not used (false). These atomic actions are easier to evaluate and some metrics, like a cost or a success probability, can be added to these leaf nodes so the tree can be assessed with quantitative analysis tools. Different types of metrics are discussed in section 1.5.

**Goal node** : The other nodes of the tree represents intermediate goals toward the main goal at the root node of the tree. Unlike leaf nodes, they are not atomic and are decomposed into other sub-nodes in further levels. These child nodes are bounded with their parent goal node through a refinement node labeled with a logic gate presented in the next section.

The basic model of attack trees admits two types of basic refinement nodes : OR and AND gates. However, many other gates [48] are proposed in the attack modeling literature. To represent the sequential behaviour of an attack scenario, the SAND (Sequential-AND) gate and other derivatives have been added to many different formal models. The SAND gate makes it possible to express a sequence in the execution of the attack steps essential to correctly define the attacker's plans.

**OR gate** : To satisfy a goal node with an OR refinement gate, at least one of its child nodes must be satisfied.

**AND gate** : To satisfy a goal node with an AND refinement gate, all of its child nodes must be satisfied.

**SAND (Sequential-AND) gate** : To satisfy a goal node with a SAND refinement gate, all of its children node must be satisfied in a specific order. This order is often represented from the leftmost child first toward the rightmost child last.

With the definition of these basic components, it is possible to build quite complete and precise attack trees. The famous example of a bank robbery has been defined 1.4 with the model we defined in this section :

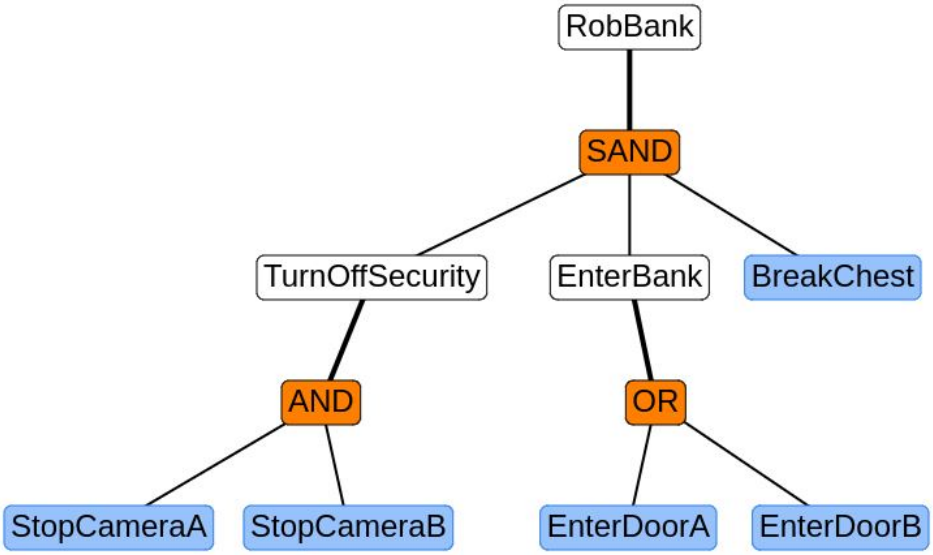



Figure 1.4: Basic theoretical attack tree to rob a bank

**Detailed explanation :** We see here the main goal of the attack "Rob bank" at the root of the tree. This is the main goal of the attacker scenario. This goal node is linked with a SAND refinement node toward 3 children nodes. This means that to achieve the goal "Rob Bank", the attacker must achieve the actions of this node's children in the "left to right" specific order. The first sub-goal is to "Turn off the security". This can be achieved by succeeding the two leaf nodes "Stop Camera A" and "Stop Camera B" because the AND gate requires it. Notice that there is no specific order to realise these two leaf nodes. The second sub-goal to achieve is to enter in the bank. The OR gate allows the attacker to only fulfill one leaf node. He can either "Enter by Door A" or "Enter by Door B". Once inside the bank, the last leaf node to succeed is to "Break the Chest" to complete the initial "Rob Bank" goal. This basic example offers a first view of two key advantages of the attack tree model :

1. It allows a clear overview of possibly complex systems. The decomposition into sub-trees makes the attack scenario more accessible to any reader looking at the specifications or weaknesses of a system.

2. Trees are scalable to bigger and more complex systems. Indeed, real devices and systems may have tens of components with tens of weaknesses depending on each other. Analysis tools using well formed tree representation can detect and highlight certain hidden threats.

This minimal model is already very expressive but can still be enhanced with small interesting features described in the two following subsections. Then, an extension of the model called Attack-defense tree will be discussed in the next section.

### Shared children

This subsection is about a small upgrade to the basic model of attack trees : the shared children. To simplify the visualization of certain trees, users may wish to use shared children to create common child nodes between multiple higher-level nodes. To allow this extension, the tree structure is enhanced toward a *directed acyclic graph* (DAG). Indeed, the structure can not be considered as a tree as cycles can be part of it. Here is the definition of DAG :

**Definition Directed Acyclic Graph [2]:** A DAG is a directed graph with no directed cycles. The edges connecting the nodes are directed and no path following the directions of these edges forms a loop.

To formally transform attack trees into DAG, we state that the directions of the edges are always facing down toward the bottom of the graph. This prevents any directed cycles as well formed trees should never have rising edges. The directions of the edge are thus not represented as it is inferred by the shape of the tree. Here is a small example using a shared leaf node needed by the two attack traces achieving the root node :

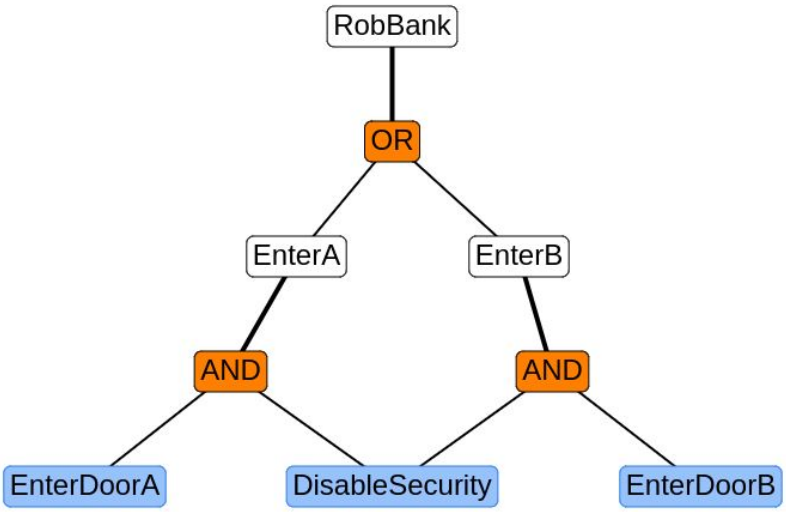



Figure 1.5: Basic shared children example

**Detailed explanation :** Two different minimal attack traces can achieve the root goal node of this simplified tree. The first one enters the bank by A and the second enters the bank by B. But both of them need to achieve the "DisalbeSecurity" leaf node. This children nodes feature has two main advantages :

1. The visual representation of the tree is simplified and no duplicated node overload the graph.

2. Shared children are a piece of supplementary useful information to analyse the attack traces of the tree. Indeed, the solvers can detect the leaf nodes used by the largest number of attacks and classify them as "central" nodes. This kind of analysis will tell the defender to use countermeasures on these critical nodes first.

### NOT statement

For certain uses, it is interesting to define negative leaf nodes and add the NOT gate between the different nodes. A logic node with a NOT gate can only be linked with 1 child node. The parent node is achieved if the child node is not. This kind of gate can be useful to create the actions of the defender on the actions of the attacker.

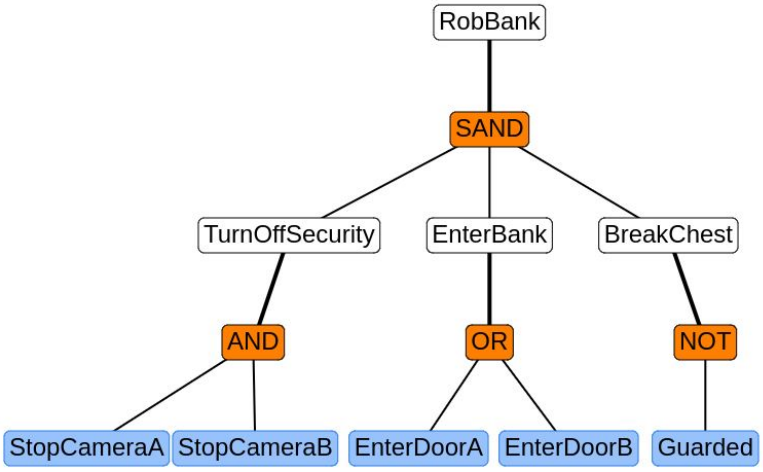



Figure 1.6: Basic NOT gate example

As this extended example shows, a NOT gate links the "BreakChest" node with the "Guarded" leaf node. The achievement of the parent node "BreakChest" requires that the "Guarded" node is not activated. It is now possible to define simple countermeasures to the actions of the attacker.

### 1.3.3 Inconsistencies

This section is a small parenthesis discussing of so possible inconsistencies. The tree structure defined in this model allows relatively broad freedom of expression but certain rules must be maintained to keep the coherence of the structure. With the addition of the two previous features, shared children and NOT gates, it is possible to design inconsistent trees. This section lists these inconsistent scenarios and the rules to apply to avoid them :

**Multiple roots :** The creation of multiple roots creates a failure as the tree is malformed. The tool can not generate a boolean formula and the exploring recursion has no unique start node.

**Loop between nodes :** Using shared children it is possible to create a loop between children and parent nodes by breaking the DAG property. The recursion loop would remain stuck forever exploring the structure. To avoid this kind of configuration, avoid a child node to be its own parent.

**One child gate :** Excluding the NOT gate, a logic gate is designed to link an output with a least two inputs. For example, and AND gate with only one input does not make any sense. This failure is less destructive than the other as the tree can still be understood and processed.

**Unsatisfying tree** : With the addition of NOT gates, it is possible to design trees accepting no solution. Again, this is not a critical failure but no analysis can be pulled out of such trees.

## 1.4 Attack-defense tree

Up to now, only the behaviour and actions of the attacker are used by the attack tree model but the measures set up by the defenders matter too. These are called countermeasures and their goal is to reduce the attackers possibilities or at least complicate them. To express the countermeasures of the defender, the model of attack tree is improved to the model called "Attack-defense tree".

**Definition attack-defense tree :** Attack–defense trees (ADTrees) enhance the expressive power of attack trees by explicitly depicting the goals of another actor, a defender, in the model [80]. In a scenario represented by an ADTree, a goal of an actor can be countered by an action of the other actor. According to the attack tree terminology, the root actor is called the proponent and the other actor is the opponent. The proponent's aim is to achieve the root goal, while the opponent tries to make it impossible. The actions of the opponent are called countermeasures. Here 1.7 is an improved example of the previous theoretical attack-defense tree with a countermeasure :

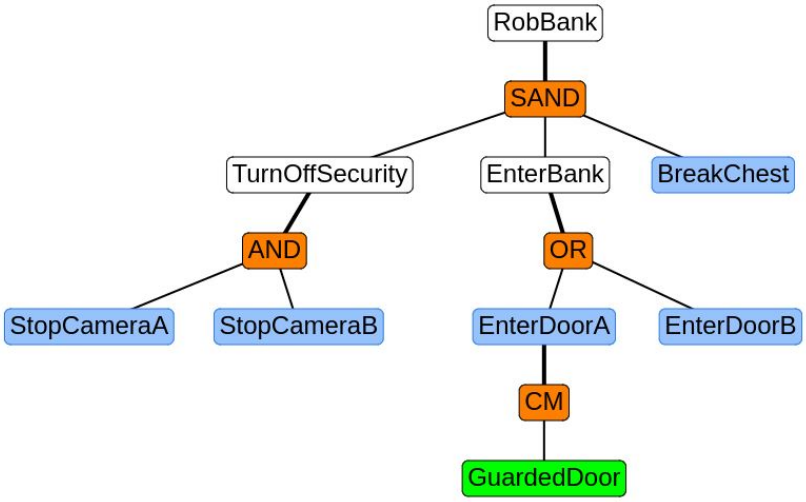



Figure 1.7: Theoretical attack tree to rob a bank with a countermeasure

In this configuration, the attack path using "EnterDoorA" is blocked by the countermeasure "GuardedDoor". This new tree is more robust than the previous one as it allows fewer possible attack paths. These additional expressions enhance the previous model towards a more complete one. The next chapter will introduce leaf metrics which can be used to compute success probabilities and costs. Countermeasures get an even better meaning while using additional attributes on the leaves of the trees.

## 1.5 Metrics

This section is related to the tree analysis of the chapter 2. Currently, the attack paths can be retrieved with SAT solvers, but the different attack paths can not really be classified. Moreover, the defenses provided by the countermeasures are not easy to quantify. The addition of different numerical metrics is an important upgrade to allow quantitative analysis. The next chapter discusses the use of a SMT solver able to interpret these metrics and classify the traces while providing a deeper analysis of the system's tree.

A document published by Mateski et al. [57] aimed at ordering a set of relevant threat metrics with a consistent threat analysis process. They state that threat metrics are anyway hard to identify, delimit, and quantify. They classify the threat metrics in two main families : *commitment* attributes that describe the threat's willingness and *resource* attributes that describe the threat's ability. Commitment attributes category is easier to evaluate as they represent quantifiable metrics. All these metrics are potential additions to the standard model of attack trees to create more precise and efficient calculations. The tool produced for this thesis is able to handle both cost and success probability attributes 4.5.

Here is a non-exhaustive list of attributes a user could add to the leaves or nodes of its system's tree to seek for a better representation of its weaknesses and a better evaluation of the treat :

**Cost :** The numerical representation of the necessary stuff required to perform that action. This can represent technical personnel, infrastructure or the acquisition of credentials. This metric can be used to determine the cheapest attack path toward the root goal of the tree.

**Success Probability :** The possibility that the attack step does work as expected. This metric allows a direct classification of the attack paths regarding their realization probability.

**Damage :** The harmful behaviour of the attack on the system or the defender. This metric can be used to determine the most critical attack path against the harmed system.

**Skill level :** The ability of the proponent to successfully achieve the attack steps. A skill level attribute could prevent unskilled attackers to be able to achieved a node. This allows counting the number of attack paths requiring low or higher skills.

**Intensity :** The diligence or persevering determination of a threat in the pursuit of its goal. Intensity is a measure of how far a threat is willing to go and what a threat is willing to risk to accomplish its goal. Attack paths with higher intensity can be considered as more dangerous.

**Stealth :** The ability of the proponent to stay unnoticed during the attack. This metric requires a certain skill level by the threat to hide from the defender but actions with higher level of stealth could allow an easier obfuscation of the attacker.

**Time :** The period of time needed to perform the action. Attack paths with a low time attribute are more likely able to succeed before any reaction from the defensive measures of the system.

The evaluation of these parameters aims to determine the critical weaknesses of the system, and to propose relevant countermeasures to the defenders. These countermeasures can be used directly in the attack tree model once their behaviour has been defined. For example, countermeasure nodes can bring an increase in cost to the protected nodes. Via quantitative analysis, users could quantify the strengths of different defense options.

## 1.6 Related work

This section enumerates different recent contributions to the domain of attack tree. Lallie et al. published in 2020 a survey [52] describing how attack graphs and attack trees represent cyber attacks in terms of their visual syntax. The graph representation offers many visual benefits but there are clear inconsistencies looking at the way real attack works. This paper provides a deep critical analysis to propose ways to standardize the visual syntax of the representations. This article also highlights that the division between the research over attack graphs and attack trees is weak as these two are essentially a graph based structure representing the same events in slightly different manners with different preconditions. A standardization between these two models would be a nice research direction.

A more general survey [36] performed by Hong et al. compared different commercial and academic prototype Graphical Security Models (GrSM) to support real-life security analysis. The document discusses about the practical issues of modeling security, about the differences between the GrSMs and their usages for larger networks. The large diversity of those models is a difficulty for the readers and this survey provides insight for these users to determine the most suitable GrSM to address their security concerns. Among these GrSM, the attack tree model is described and links this survey with the theme of this thesis.

The last two surveys directly concern attack-tree based security models. The first one published by Kordy et al. [48] in 2014 resumed the state-of-the-art around the DAG-based security modeling until 2013. The global observation expose 2 general trends regarding the field of graphical security modeling : *unification* and *specification*. The objective of the first trend is to unify the actual models to propose a general one useful to analyse a wide variety of security scenarios and security systems. The specification trend aims at the creation of models responding to precise security cases. These are often based on practical needs and focuses on domain-specific metrics. Finally, the document proposes some orientations for the development of new models like the focus on general attacks or vulnerability patterns.

The second survey directed by Wide et al. [80] in 2019 followed up this previous research but mainly considers attack trees and their extensions and attack-defense trees adding countermeasure nodes to their model. The goal of the survey is only to classify the different available features from any model and present them in a unified way. They also explore the researches around automated attack tree generation and the addition of sequential nodes to deal with dependencies between the different actions. These two topics are the main contribution of this thesis and implement a way to use these features.

# Chapter 2

# Attack tree analysis

## 2.1 Introduction

Now that the state-of-the-art around attack tree models has been covered this chapter will focus on the analysis of such trees. Once the tree closely represents the system, users want to perform quantitative analysis to highlight the attack traces and classify them. The goal of this chapter is to list and clarify all the steps and tools used for boolean quantitative analysis. The first step is to translate the tree into a logical sentence formed by the leaves and the structure of the tree. Then, solvers will use this sentence to compute all the configurations which solve the tree. These solutions, called attack paths, are the key points to understanding the weaknesses of the studied system. This chapter summarizes the basis of boolean solving and applies these procedures to the solving of attack trees.

## 2.2 Boolean algebra

Boolean algebra is the branch of mathematics which regroups the algebra of truth values and their operations. Truth values are the Boolean values True and False which are binary variables often denoted as T (or 1) and F (or 0). It is deeply connected to Propositional logic and correspondences between both can be made with minor changes in notation and terminology. Its main usage is the description of logical operations. These are operations which take truth values as inputs to produce a single output which is also either True or False.

The basic Boolean operations (with their corresponding operators) are conjunction (AND), disjunction (OR) and negation (NOT). They can take any number of inputs (operands) except for the negation which uses a unary operator (operation with only one operand/input). Additionally, it is useful to define the parity (XOR), material implication (IMPLIES) and material equivalence (IF AND ONLY IF

or IFF). All these operations can be combined together to form what we call Boolean formulas or expressions on Boolean variables (or Propositions on Atoms in propositional calculus). It is also important to note that the basic operator precedence is (from highest to lowest) : parenthesis, negations, conjunctions and finally disjunctions.

### 2.2.1 Boolean operations

The following table lists the symbols describing the operators used in this document and implemented into the tool (as there is no consensus in logical notation, multiple symbols for each operation exist across articles and libraries).

| Operator | Function | Theoretical Notation | Implementation Notation |
|---|---|---|---|
| NOT | Is False | $\neg, \overline{\ }$ | $\sim, !$ |
| AND | All True | $\wedge$ | & |
| OR | At least one True | $\vee$ | \| |
| XOR | # of True is odd | $\oplus$ | ^ |
| IMPLIES | Not Left Or Right | $\rightarrow$ | Not used |
| IFF | All True Or All False | $\leftrightarrow$ | Not used |

The main operations and their respective negations are described below with their truth tables for 2 input variables inside tabular representations.

| | NOT |
|---|---|
| | negation |
| A | $\neg A$ |
| F | T |
| T | F |

| | | AND | NAND | OR | NOR |
|---|---|---|---|---|---|
| | | conjunction | $\neg$conjunction | disjunction | $\neg$conjunction |
| A | B | $A \wedge B$ | $\neg (A \wedge B)$ | $A \vee B$ | $\neg (A \vee B)$ |
| F | F | F | T | F | T |
| F | T | F | T | T | F |
| T | F | F | T | T | F |
| T | T | T | F | T | F |

All the other Boolean operations can be built by combination of these primary operations (method used to combine operations). The following table presents notable examples of such operations :

| | | XOR | XNOR | IMPLIES | IFF |
|---|---|---|---|---|---|
| | | parity | ¬parity | implication | equivalence |
| A | B | $A \oplus B$ | $\neg (A \oplus B)$ | $A \rightarrow B$ | $A \leftrightarrow B$ |
| F | F | F | T | T | T |
| F | T | T | F | T | F |
| T | F | T | F | F | F |
| T | T | F | T | T | T |

### 2.2.2 Logical equivalence

In mathematical logic two statements are said to be **logically equivalent** if, and only if, they have the same truth table for every input combination. The symbol used to represent logical equivalence is ≡ and is defined along with another logic statement, the tautology. The tautology is the set of all logical propositions that always evaluate to True for each possible interpretation. With this additional statement, the logical equivalence can be defined as follows : Two logical formulas p and q are logically equivalent (denoted p ≡ q) if and only if p ↔ q is a tautology.

Some of the most commonly used logical equivalences, which are often listed as laws or properties, are illustrated in the following table.

| | |
|---|---|
| $p \wedge True \equiv p$<br>$p \vee False \equiv p$ | Identity laws |
| $p \vee True \equiv True$<br>$p \wedge False \equiv False$ | Domination laws |
| $p \vee p \equiv p$<br>$p \wedge p \equiv p$ | Idempotent or tautology laws |
| $\neg(\neg p) \equiv p$ | Double negation law |
| $p \vee q \equiv q \vee p$<br>$p \wedge q \equiv q \wedge p$ | Commutative laws |
| $(p \vee q) \vee r \equiv p \vee (q \vee r)$<br>$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ | Associative laws |
| $p \vee (q \wedge r) \equiv (p \vee q) \wedge (q \vee r)$<br>$p \wedge (q \vee r) \equiv (p \wedge q) \vee (q \wedge r)$ | Distributive laws |
| $\neg(p \wedge q) \equiv \neg p \vee \neg q$<br>$\neg(p \vee q) \equiv \neg p \wedge \neg q$ | De Morgan's laws |
| $p \vee (p \wedge q) \equiv p$<br>$p \wedge (p \vee q) \equiv p$ | Absorption laws |
| $p \vee \neg p \equiv True$<br>$p \wedge \neg p \equiv False$ | Negation laws |
| $p \rightarrow q \equiv \neg p \vee q$ | Implication equivalence |
| $p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$ | Bi-Implication equivalence |

## 2.3 Boolean satisfiability problem

The Boolean satisfiability problem (also called SAT) is the problem of determining whether there exists a combination of inputs truth values which makes a propositional formula evaluates to True. A formula evaluating to True for at least one variable assignment is called satisfiable. Otherwise, if every variable assignment evaluates the formula to false, this formula is called unsatisfiable.

■ *example*

| | | Satisfiable Formula | Unsatisfiable Formula |
|---|---|---|---|
| A | B | $A \wedge \neg B$ | $A \wedge \neg A$ |
| F | F | F | F |
| F | T | F | F |
| T | F | T | F |
| T | T | F | F |

SAT is the first problem which was proved to be **NP-Complete** (Non-deterministic Polynomial-time complete) by Stephen A. Cook in 1971 [21] (Cook-Levin theorem). No efficient solution algorithm has been found to solve this class of problems. Two properties define this class of problem :

- (**NP** set) Their solutions can be verified in polynomial-time and a brute-force algorithm is needed to find a solution as no particular rule is followed to make the guess.
- (**NP-Complete** set) A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time.

The **NP** set comes along with the **P** problem class, the set of problems that can be solved by Deterministic Polynomial-time Turing Machine. By definition, the P set is included in NP. These elements form the basis of the P versus NP problem. This problem asks whether every problem with solutions verifiable in polynomial time can be solved in polynomial time too. At this time, the answer to this problem is uncertain and we cannot say if $P = NP$ or if it is unprovable.

This also means that finding a polynomial time algorithm to verify the satisfiability of a boolean formula is quite unlikely and that the computational complexity of the best algorithm might have an exponential time behavior in the worst case as this problem is NP-complete. However SAT algorithms are used in many practical applications (as explained in Marques-Silva paper [56]) such as model checking, combinational equivalence checking, planning in artificial intelligence... These applications show that there is a great interest in the implementation of fast and practical tools capable of solving the SAT problem for a given instance. Such tools are called SAT Solvers.

### 2.3.1 Normal Forms

In propositional logic every formula can be converted into an equivalent formula following a specific set of rules by using logical equivalences. Among these forms there is a need for a standard format using a small set of operators to ease the satisfiability analysis of such formula. These representations are called Normal Forms and this section exposes three of them restricted to the use of the conjunction, disjunction and negation operators ($\wedge, \vee$ *and* $\neg$).

**Literals :** Literals of a formula are the atomic variables or their negations composing that formula.

- **Negation Normal Form (NNF)** :
  Negation operator is only applied to the variables
- **Disjunctive Normal Form (DNF)** :
  Disjunction of minterms which are conjunctions of literals (without duplicates)

$$\bigvee_i \left( \bigwedge_j l_{i,j} \right)$$

- **Conjunctive Normal Form (CNF)** :
  Conjunctions of clauses (or maxterms) which are disjunctions of literals (without duplicates)

$$\bigwedge_i \left( \bigvee_j l_{i,j} \right)$$

with $l_{i,j}$ the j-th literal of the i-th term in both cases

Now that the composition of each normal form has been described, which one should be used to determine the satisfiability of a given SAT problem ?

NNF could be used but is not very useful as additional computations are needed to overcome the greater number of logical layers compared to CNF and DNF. The satifiability of a DNF formula can easily be solved in P as we can simply iterate over all the conjunctions to find the solutions by checking their respective satifiability (the total set of solutions is the union of the set of solutions of each conjunction). This is why CNF is sometimes considered as implicit whereas DNF is explicit. So we know that DNF-SAT is efficiently solvable, but why do most solvers use CNF (with a NP-complete satifiability) as input and CNF-SAT is often the default solving process in implementations ?

The bottleneck of DNF is the computational cost compared to CNF. The conversions of the two forms are relative to the size of the initial formula. Both can have an exponential size explosion of during the conversion but CNF can be shortened efficiently by introducing new variables to replace occurrences of subformulas (it is thus possible to quickly find an equisatisfiable CNF). As well, it is easier to combine two CNF formulas (which makes it easy to remember partial assignments) and to detect conflicts. This linear time and space transformation and the possibility of pruning the search space (speed up the search) are the main reasons why this approach is widely used in practice.

### 2.3.2 CNF transformation

There is many ways to transform a boolean formula into CNF. Three different transformations will be studied in this section and their performances will be compared. As a quick reminder, our attack tree tool will chain a CNF transformation and a SAT solving procedure to assess the solutions of the attacks. It is attractive to select the most effective computation method as the size of attack trees can be quite large.

#### Conversion with the propositional logic

1. Conversion to negation normal form :
   - Replace XOR, IMPLIES or IFF relations to AND, OR, NOT clauses using logical equivalence
   - Move NOTs inwards by repeatedly applying De Morgan's Law (negations only apply to atoms)
   - Eliminate double negations.

   ■ *example* : $R = (B \wedge (A \rightarrow (A \wedge B))) \vee \neg(A \vee B)$

   $$\boxed{\text{Negation normal form:}}$$
   $$\text{Replace } \oplus, \rightarrow, \leftrightarrow$$
   $$A \rightarrow (A \wedge B) \equiv \neg A \vee (A \wedge B) \quad \Rightarrow \quad (B \wedge (\neg A \vee (A \wedge B))) \vee \neg(A \vee B)$$
   $$\text{Move negation inwards and eliminate double negations}$$
   $$\neg(A \vee B) \equiv \neg A \wedge \neg B \qquad \Rightarrow \quad (B \wedge (\neg A \vee (A \wedge B))) \vee (\neg A \wedge \neg B)$$

2. Distribute ORs inwards over ANDs repeatedly by applying the distributive law where a disjunction occurs over a conjunction. Once this is not possible anymore, it means that the resulting formula is in CNF.

■ *example* :

| | Distributive law of ORs over ANDs | |
|---|---|---|
| (0) | $(B \wedge (\neg A \vee (A \wedge B))) \vee (\neg A \wedge \neg B)$ | (1) $\wedge$ (6) |
| (1) | $(B \wedge (\neg A \vee (A \wedge B))) \vee \neg A$ | (2) $\wedge$ (3) |
| (2) | $B \vee \neg A$ | (2) is final |
| (3) | $(A \wedge B) \vee \neg A$ | (4) $\wedge$ (5) |
| (4) | $A \vee \neg A$ | (4) is final |
| (5) | $B \vee \neg A$ | (5) is final |
| (6) | $(B \wedge (\neg A \vee (A \wedge B))) \vee \neg B$ | (7) $\wedge$ (8) |
| (7) | $B \vee \neg B$ | (7) is final |
| (8) | $\neg A \vee (A \wedge B) \vee \neg B$ | (9) $\wedge$ (10) |
| (9) | $\neg A \vee A \vee \neg B$ | (9) is final |
| (10) | $\neg A \vee B \vee \neg B$ | (10) is final |

Without repetition the final CNF expression is :

$$R = (B \vee \neg A) \wedge (A \vee \neg A) \wedge (B \vee \neg B) \wedge (\neg A \vee A \vee \neg B) \wedge (\neg A \vee B \vee \neg B)$$

Which could be simplified further with the Negation and Domination laws :

$$R = B \vee \neg A$$

\+ Advantages :

- Easy recursive computation.

– Disadvantages :

- Potential creation of exponentially increasing number of clauses due to applications of the distributive law.

### Quine–McCluskey transformation

The Quine–McCluskey algorithm (QMC, [59]) (also called the method of primer implicant or the tabulation method) is a method used to minimize Boolean functions. It is functionally similar to Karnaugh mapping (K-Map, [44]) but it uses a tabular form which makes it more efficient for computer algorithms. It is better than K-Map for expressions with more than 4 input variables and it gives a deterministic way to check if the minimal form of a boolean function has been found. This method is performed in two main steps :

1. Finding the prime implicants of the function :
   We use the inputs evaluating the formula to True (also called minterms) found with the truth table of the Boolean expression to create a minterm table and find the implicants of the formula. Each midterm becomes an implicant of size 1 and we try to combine them two by two to simplify the table. When a term (implicant of size $\geq 1$) can't be combined anymore it is considered as a prime implicant of the expression.

   ■ *example* : $R = (B \wedge ((A \wedge B) \vee (\neg A))) \vee (\neg A \wedge \neg B) = \sum m(0, 1, 3)$

Truth table and implicants:

Truth Table

| | A | B | R |
|---|---|---|---|
| 0 | F | F | T |
| 1 | F | T | T |
| 2 | T | F | F |
| 3 | T | T | T |

$\Rightarrow$

Implicants (size 1)

| | A | B |
|---|---|---|
| 0 | F | F |
| 1 | F | T |
| 3 | T | T |

$\Rightarrow$

Implicants (size 2)

| | A | B |
|---|---|---|
| 0,1 | F | - |
| 1,3 | - | T |

Prime implicants

0,1 : $\neg A$
1,3 : $B$

2. Use them to find the essential prime implicants of the function necessary to cover the whole function : To achieve this we have to make sure that all the True clauses are covered by the implicants. We do this by creating the prime implicant chart to see which minterms are covered. When a column of the chart has a single prime implicant it is called an essential prime implicant of the solution and is needed in the minimized boolean equation. Every minterms must be covered by the chosen implicants forming the minimal clauses combination shortening the initial expression.

   ■ *example* :

Prime implicant chart:

With coverage

| | A | B | 0 | 1 | 3 |
|---|---|---|---|---|---|
| 0,1 | F | - | ○ | ○ | |
| 1,3 | - | T | | ○ | ○ |

$\Rightarrow$

With essential

| | A | B | 0 | 1 | 3 |
|---|---|---|---|---|---|
| 0,1 | F | - | ● | ○ | |
| 1,3 | - | T | | ○ | ● |

$\Rightarrow$

Minimized expression

$R = \neg A \vee B$

Sadly this technique uses the truth table of the Boolean function to compute the minimal CNF formula. This means that it can't be used in our case as it needs to try each $2^n$ possibilities of the expression to find which ones should be used in the reduction. The only way to use this technique efficiently would have been to get an easy-to-compute DNF conversion to find the minterms and to use this algorithm to minimize its clauses but such technique has yet to be found. Note

that this technique computing the prime implicates with a truth table does the same computations as the SAT solver and might totally prevent its usage if the expression directly simplifies to a boolean value.

+ Advantages :

  - Returns a minimal CNF or DNF expression.

− Disadvantages :

  - Uses an internal truth table to find the prime applicant (which is redundant with the solver's duty and quite computationally intensive).

**Tseitin transformation**

This third transformation introduced by GS Tseitin in [78] really makes CNF-SAT an easier problem to compute. Its main concept is the possibility to produce an equisatisfiable CNF formula from an arbitrary logical expression while keeping linear conversion size. This is achieved by introducing new variables for sub-formulas (excluding existing variables and by using various logic gates transformations which will be developed later) and enforcing their satisfaction (by adding a final literal to the conjuntion to force the expression output to True).

**Equisatisfiability :** The equisatisfiability between two boolean formulas is achieved if, and only if, both formulas are satisfied whenever one of them is.

Generally, any CNF transformation achieves equivalence between the input formula and the transformed one. However, the Tseitin conversion only achieves equisatisfiability because of the additional variables in the transformed formula. The basis of Tseitin conversion is given below :

$$\phi = \textit{Boolean formula}$$
$$\forall \text{ sub-formulas f : we define } a_f, \text{ the corresponding auxiliary variable}$$

$$T(\phi) = \underbrace{a_\phi}_{\text{Final Clause}} \bigwedge_f \underbrace{(a_f \leftrightarrow \textit{Formula } f \textit{ with Substitutions})}_{\textit{Sub-expressions}}$$

Each of these sub-expressions can be quickly developed into CNF by applying basic conversion or recursion on $a_f \leftrightarrow (a_g \text{ operator } a_h)$. We use seven main operations as default sub-expressions to be simplified. Technically every operation could be converted but it is often simpler to restrict the conversion to the most used operators. The CNF results for each operation are listed for 2 inputs (except for the unary operator) in figure 2.1.

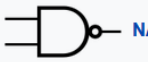
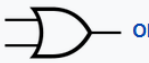
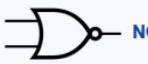
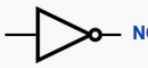
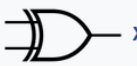
| Type | Operation | CNF Sub-expression |
|---|---|---|
|  AND | $C = A \cdot B$ | $(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$ |
|  NAND | $C = \overline{A \cdot B}$ | $(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee C) \wedge (B \vee C)$ |
|  OR | $C = A + B$ | $(A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C)$ |
|  NOR | $C = \overline{A + B}$ | $(A \vee B \vee C) \wedge (\overline{A} \vee \overline{C}) \wedge (\overline{B} \vee \overline{C})$ |
|  NOT | $C = \overline{A}$ | $(\overline{A} \vee \overline{C}) \wedge (A \vee C)$ |
|  XOR | $C = A \oplus B$ | $(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$ |
|  XNOR | $C = \overline{A \oplus B}$ | $(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee B \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee B \vee \overline{C})$ |

Figure 2.1: CNF Sub-expressions with 2 inputs

With these transformations operations for 2-inputs gates becomes effortless. But in our case it would be interesting to be able to analyze boolean operations with any number of inputs. It could be possible to use recursion and introduce new variables for each additional operand but it would worsen the solving process. Instead, for most logical operations, a more adequate generalized CNF sub-expression for n inputs can be derived from basic conversion :

- **NOT** : (no change as it is a unary operator)
  - ▶ negation function
    $$f = \neg x$$
  - ▶ negation transformation (2 clauses)
    $$(\neg f \vee \neg x) \wedge (f \vee x)$$
- **AND** :
  - ▶ conjunction function
    $$f = \bigwedge_{i=0}^{n-1} x_i$$
  - ▶ conjunction transformation (n+1 clauses)
    $$\left(f \vee \bigvee_{i=0}^{n-1} \neg x_i\right) \wedge \bigwedge_{i=0}^{n-1} (x_i \vee \neg f)$$

- ▶ **NAND** : replace f by ¬f in conjunction transformation

- **OR** :
  - ▶ disjunction function
  $$f = \bigvee_{i=0}^{n-1} x_i$$
  - ▶ disjunction transformation (n+1 clauses)
  $$\left(\neg f \vee \bigvee_{i=0}^{n-1} x_i\right) \wedge \bigwedge_{i=0}^{n-1} (\neg x_i \vee f)$$
  - ▶ **NOR** : replace f by ¬f in disjunction transformation

- **XOR** : (in order to show that we have to try every combination of negated or non negated variables we used : V as the set of variables, K as a subset of variables to negate and L as the subset of the remaining variables)
  - ▶ parity function
  $$f = \bigoplus_{i=0}^{n-1} x_i = \bigwedge_{\substack{K \subseteq V,\ L = V-K \\ |L| \text{ is even}}} \left(\bigvee_{i \in K} x_i \vee \bigvee_{j \in L} \neg x_j\right)$$
  - ▶ parity transformation ($2^n$ clauses)
  $$\bigwedge_{K \subseteq V,\ L = V-K} \left(F(L) \vee \left(\bigvee_{i \in K} x_i \vee \bigvee_{j \in L} \neg x_j\right)\right)$$
  with :
  $$V = \{x_0, x_1, ..., x_{n-1}\}$$
  $$F(L) = \begin{cases} \neg f & \text{if (n and } |K| \text{ are even) or (n and } |K| \text{ are odd)} \\ f & \text{otherwise} \end{cases}$$
  - ▶ **XNOR** : replace f by ¬f in parity transformation

  In this case, the resulting formula gets an exponential blown-up in the number of clauses. It is still possible to introduce supplementary variables and use recursion to reduce the number of clauses (when n $\geq$ 2) from $2^n$ to $4(n-1)$ by using :
  $$\bigwedge_{i=2}^{n} \Big((\neg T_{i-1} \vee \neg x_{i-1} \vee \neg T_i) \wedge (T_{i-1} \vee x_{i-1} \vee T_i) \wedge (T_{i-1} \vee \neg x_{i-1} \vee \neg T_i) \wedge (\neg T_{i-1} \vee x_{i-1} \vee \neg T_i)\Big) \qquad \text{with } T_1 = x_0$$

This is done by computing the Tseitin transformation of the XOR formula 2 by 2 as :

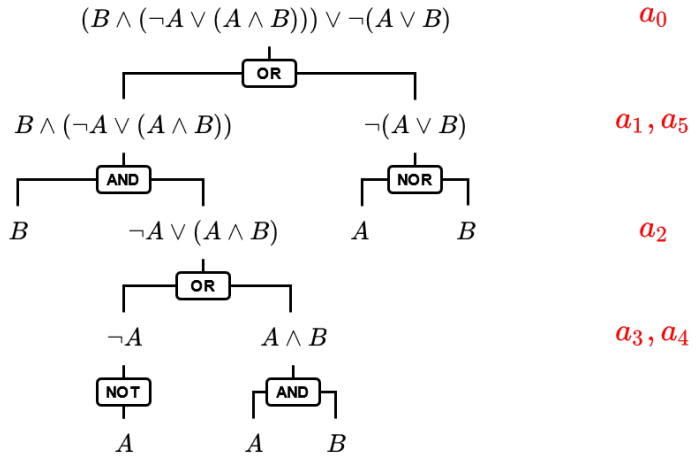
$$x_0 \oplus x_1 \oplus ... \oplus x_{n-1} = (((x_0 \oplus x_1) \oplus ...) \oplus x_{n-1})$$

$$T_n \wedge (T_n \leftrightarrow (x_0 \oplus x_1 \oplus ... \oplus x_{n-1}) \equiv T_n \wedge (T_2 \leftrightarrow (x_0 \oplus x_1)) \wedge \bigwedge_{i=3}^{n} (T_i \leftrightarrow (T_{i-1} \oplus x_{i-1}))$$

The main trade-off remains the same as we need to increase the number of variables to decrease the number of clauses.

■ *example* :

Auxiliary Variables introduction:




Equisatisfiable formula with sub-expressions convertion:

$$T(\phi) = a_0 \ \wedge (a_0 \leftrightarrow a_1 \vee a_5) \wedge (a_1 \leftrightarrow B \wedge a_2) \wedge (a_5 \leftrightarrow \neg(A \vee B)) \wedge$$
$$(a_2 \leftrightarrow a_3 \vee a_4) \wedge (a_3 \leftrightarrow \neg A) \wedge (a_4 \leftrightarrow A \wedge B)$$

Then we just need to use the adequate CNF transformation for each clause with auxiliary variables to get the complete formula in CNF.
$a_0 \leftrightarrow a_1 \vee a_5$ becomes $(\neg a_1 \vee \neg a_5 \vee a_0) \wedge (a_1 \vee \neg a_0) \wedge (a_5 \vee \neg a_0)$
...
Once solved, we can just discard the auxiliary variables from the final solution.

| + Advantages : | − Disadvantages : |
|---|---|
| • Linear time and space complexities.<br>• Practical to implementation. | • Not always efficient for small boolean formulas.<br>• Auxiliary variables expand the search space (which might increase solving difficulty). |

### Comparison

This comparison will use the two relevant types of CNF conversion (basic and Tseitin's) and try to solve them with the same SAT-solver. This will show the impact of different encodings on the solving time to determine which one is more suited to the computation of an attack tree. We compared the results of both encoding to the result of a simple truth table method (which is essentially a brute-force algorithm) and looked for the computation time evolution regarding the number of variables and the complexity of the boolean expressions. We also used a 5 minutes timeout along with a limitation on the number of solutions (10.000) to reduce the impact of unstable computations. It is also important to note that while each conversion implementation is made in *Python*, the SAT solver comes from a *C++* library and the truth table processing is also implemented with *Python* (this should influence the global solving speed but still allow a fair comparison). The results (measured in seconds) are given below :

| | Truth Table | | | Basic | | | Tseitin | | |
|---|---|---|---|---|---|---|---|---|---|
| | Encode | Solve | Total | Encode | Solve | Total | Encode | Solve | Total |
| T1 | 0.0 | 0.010 | 0.010 | 0.010 | 0.001 | 0.012 | 0.018 | 0.002 | 0.020 |
| T2 | 0.0 | 0.037 | 0.037 | 1.967 | 0.005 | 1.972 | 0.181 | 0.013 | 0.194 |
| T3 | 0.0 | 1.932 | 1.932 | 0.627 | 0.175 | 0.802 | 0.076 | 0.220 | 0.296 |
| T4 | 0.0 | 266.831 | 266.831 | 92.852 | 1.304 | 94.156 | 0.658 | 1.201 | 1.859 |

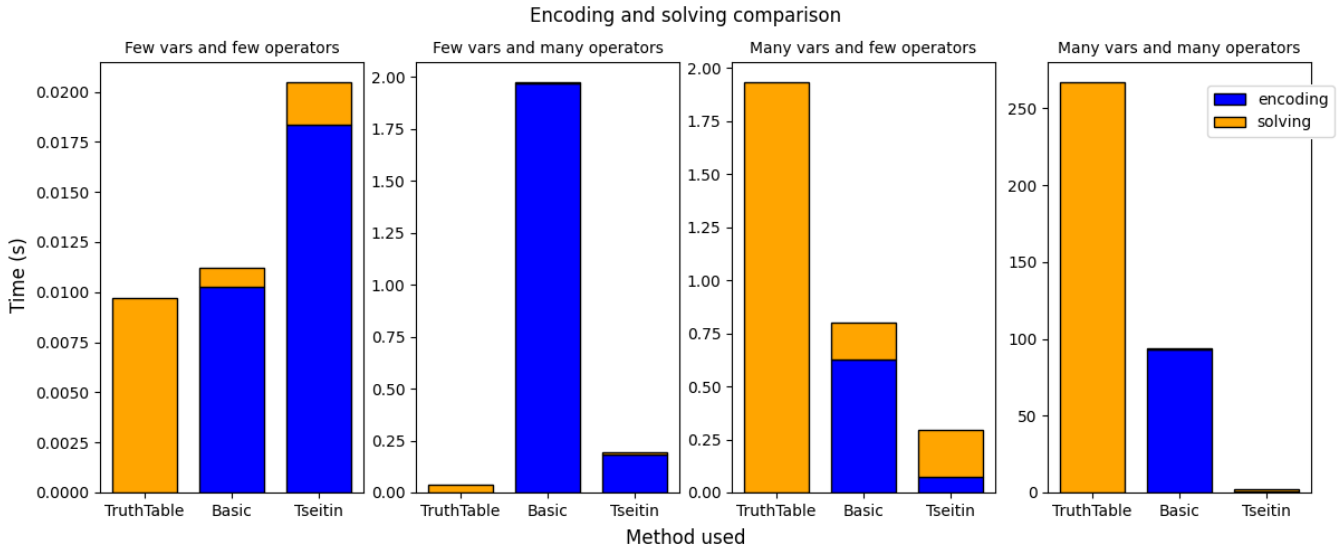



Figure 2.2: Instances testing

Figure 2.2 illustrates that, for a few variables to iterate on, the truth table achieves the computations faster. This is expected as CNF conversion consumes a lot of resources to get a result that still needs to be solved (this is even more obvious when many operators are slowing the conversion). The graphs also show that Tseitin transformation is the best way to find solutions for more complex formulas and that the loss in solving time is greatly compensated by the gain in encoding time compared to basic conversion. In the case of Attack Tree, the formula often falls into the third and fourth categories. This makes Tseitin the evident choice to optimize our implementation and find solutions with CNF-SAT solvers.

### 2.3.3 SAT solvers

The next step in our analysis is the understanding of the way SAT solving programs work, in particular the CNF-SAT solvers as they are the most common implementations. As previously stated, Boolean satisfiability is supposedly limited by algorithms with average exponential worst-case complexity. Despite this, many algorithms have been developed and optimized for efficiency and scalability using heuristics and program optimizations. Multiple approaches were considered to get better results such as restricting the input form to get polynomial-time situations or experimenting with the formula parameters like the ratio of the number of clauses to the number of variables. But the behaviour of every SAT solvers is greatly dependent on the instance given. They all come with pros and cons (some act as specialized tools) and can't be good for every practical application. Let's describe the most commonly used SAT solving implementations :

#### Stochastic algorithms

Stochastic Local Search algorithms (SLS) work by assigning a random value to each variable and checking if all clauses are satisfied. If not, a variable is flipped and the process is repeated. A solution is found if all clauses are satisfied. It may also restart with random values after too many fails to be able to get out of local minimas. The main differences between implementations are the methods used to select the variable to modify after a failure. We can cite WalkSAT and GSAT as solvers using this approach. Sadly there is no guarantee that these techniques will find a solution even if it exists and they also can't show that no solution exists (the algorithm is called incomplete as it can loop and never terminate).

Another notable technique making use of randomness and heuristics is the PPSZ algorithm (from R Paturi, P Pudlák, ME Saks and F Zane paper [63]) which uses a preprocessing stage and then tries to incrementally construct a satisfying assignment

by choosing random variables. It is currently the fastest known algorithm for the k-SAT problem (where each clause has at most k literals).

### DPLL algorithm

The Davis–Putnam–Logemann–Loveland (DPLL, [24]) algorithm is a complete algorithm based on backtracking and searching methods to solve the CNF-SAT problem. This method improves the resolution-based Davis-Putnam (DP, [25]) algorithm by getting rid of its potentially exponential memory consumption.

$$\textit{Resolution rule simplification} : \quad \frac{(X \vee A) \quad , \quad (\neg X \vee B)}{(A \vee B)}$$

The extended method works as follows : (1) it iterates over the clauses, (2) selects a variable, (3) assigns it a truth value, (4) simplifies the formula and finally (5) checks its satisfiability. The iteration process stops when a satisfying assignment is found or when a conflict arises. In this case, the process backtracks and tries another value assignment. If every value assignment have been explored, the formula is unsatisfiable. A conflict in the current assignment can be identified when every literals in a clause have been set to False as the CNF form is a conjunction of disjunctive clauses. To enhance the process, the algorithm uses additional rules at each step :

- *Unit propagation* : If a clause is a unit clause (contains only one unassigned literal), the remaining literal must be set to True, enabling simplification in the rest of the formula.
  - *example* :

| | |
|---|---|
| $(False \vee X) \wedge (X \vee B) \wedge (\neg X \vee A) \wedge (\neg A \vee B)$ | X must be True |
| $(\neg X \vee A) \wedge (\neg A \vee B)$ | Clauses with X can be removed |
| $A \wedge (\neg A \vee B)$ | $\neg X$ can be removed from clauses |

- *Pure literal elimination* : If a literal only occurs with one polarity in the CNF formula (all positives or all negatives) it is called pure and can make every clause containing it True meaning that they can be deleted.
  - *example* :

| | |
|---|---|
| $(X \vee A) \wedge (X \vee \neg B) \wedge (\neg Y \vee A) \wedge (\neg A \vee B)$ | X and Y are pure (X : True, Y : False) |
| $(\neg Y \vee A) \wedge (\neg A \vee B)$ | Clauses with X can be removed |
| $(\neg A \vee B)$ | Clauses with $\neg$Y can be removed |

In is important to note that the chosen order of variables picked when iterating has a significant impact on the performance of the solver. Many heuristics were developed to overcome or take advantage of this fact. For example, some of them assign scores to variables at runtime according to the number of conflicts they were into while others evaluate this score according to the number of clauses a literal would satisfy. These heuristics are also often processed along with other techniques like random restarts, preprocessing, adaptive branching or clauses deletion. Nowadays the DPLL algorithm is used as a basis for most successful solvers.

#### CDCL algorithm

One important evolution of the DPLL algorithm is the Conflict-Driven Clause Learning (CDCL) algorithm using many additional techniques along with the DPLL approach. The main difference resides in the introduction of non-chronological backtracking (or backjumping) which uses the conflict clauses (negation of the assignments that led to the conflict) to guide the search. By finding the earliest variable assignment involved in the conflict and adding conflict clauses to the problem, the solver is able to avoid repetitions of the same conflict (by essentially pruning part of the search tree).

#### Parallel solving

An interesting method of parallel solving comes from the previously stated fact that no solver can be the best at solving every problem. It is very difficult to predict which algorithm will perform better on a given instance and these limitations motivated the creation of Parallel Portfolio methods. These methods concurrently run multiple solvers or configurations on the tested instances to increase the chance to solve them efficiently.

## 2.4 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem is a decision problem which generalizes the SAT problem to more complex formulas using a richer modelling language. Determining the satisfiability of a mathematical formula remains the main goal but whereas SAT is limited to propositional logic (using boolean variables), SMT problems accept first-order logic (using predicates as atomic formulas) to add expressions involving quantifiers (not always because of their computation difficulties and by using finite domains or relevant quantifiers elimination techniques), numbers, data structures, functions, non-logical symbols, constants, ... As boolean SAT

problems are NP-complete, SMT problems are at least as hard to solve (NP-hard). They are also completely undecidable for many theories.

### 2.4.1 SMT solvers

For this kind of problem diverse approaches exist and are used in many implementations of SMT solvers. It is incorrect to say that SMT solvers are more powerful than SAT solvers. They will share the same exponential runtime and incomplete behaviour when solving problems. The advantage of SMT solvers comes from their higher level of abstraction, which facilitates the expression of higher-level constraints (with associated background theories). This allows them to be faster when proving more complex statements with "obvious" facts (such as $x + y = y + x$ in integer addition) compared to cheaper boolean SAT solvers. They have many applications in hardware verification, security and testing.

Most SMT solvers follow the **DPLL(T) framework** (figure 2.3) which is divided into two interacting parts :

1. A **SAT solver with DPLL-based search** to explore the boolean formula.
2. **Theory-specific solvers** (T-solvers) to check the feasibility of conjunctions of theory predicates given by the SAT solver and gain knowledge from them when conflicts arise.

For this architecture to work efficiently, the theory solver has to participate actively in the propagation and conflict analysis by using previously gathered information. Furthermore, it has to give insights to the SAT solver in case of infeasibility caused by theory conflicts.

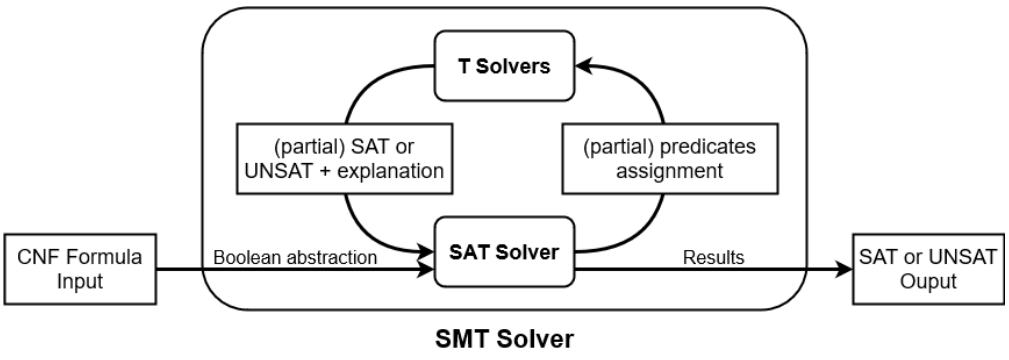



Figure 2.3: DPLL(T) framework design

### 2.4.2 Theories of interest

Many SMT solvers work with multiple theories at the same time and are needed by the solver theories to evaluate expressions. Each theory has sub-logic with specific processing methods and uses them whenever it is necessary. Some of the most relevant are listed here :

- Equality and Uninterpreted Functions (EUF) : basic part of first-order predicate logic (constants, functions, predicate symbols, equality)
- Arithmetic : linear or non-linear, integer or real values
- Arrays : to model actual array data structure without needing the whole original array memory size
- Bit-Vectors : compact representation, floating point arithmetic, bit-wise operations
- Algebraic Datatypes : theory of finite trees and other structures
- Sequences and Strings

### 2.4.3 Adaptation to Attack Tree

The next step is to clarify the use of SMT solver to attack tree solving. Whereas SAT solving gives the boolean solution to a given formula representing an Attack Tree, it is also useful to be able to assign numerical metrics 1.5 to the boolean variables representing the leaves of the tree. The abilities of SMT solvers allow computation under specific numerical conditions or particular assigned parameters. This is why a SMT solver can be used to find the combination of variables enabling the root node of an Attack Tree with the minimum cost or the maximum success probability. This can be achieved by combining the initial boolean formula with a cost/probability calculation function along with an upper/lower bound condition. To avoid incorrect computations, a domain condition has to be added for costs (condition : greater than 0) and for probabilities (condition : between 0 and 1). These domains can be inserted to the formula by creating a conjunction between their conditions. Therefore the SMT input expression used for these two calculations formulates as follows :

$$\phi_{SMT} = Domain \wedge Boolean\ Formula \wedge (Result == Evaluation) \wedge Bounding$$

The *Result* variable gives the return value of the evaluation function and the bounding condition restricts the search or gets all the solutions within an interval. The terms of the two cases are defined below :

**Cost of an attack**

$$C(x) : \text{the assigned cost of variable } x$$

*Domain* :

$$C(x_i) \geq 0 \quad \forall\, i \in \{0, 1, ..., n-1\}$$

*Cost evaluation function* : (sum of costs from taken variables)

$$C(x_0, x_1, ..., x_{n-1}) = \sum_{i=0}^{n-1} c(x_i)$$

$$\text{with } c(x) = \begin{cases} C(x) & \text{if x is True} \\ 0 & \text{otherwise} \end{cases}$$

*Bounding Condition* :

$$Result \leq Cost\ Upper\ Bound$$

**Probability of an attack**

$$P(x) : \text{the assigned probability of variable } x$$

*Domain* :

$$0 \leq P(x_i) \leq 1 \quad \forall\, i \in \{0, 1, ..., n-1\}$$

*Probability evaluation function* : (product of probabilities from taken variables)

$$P(x_0, x_1, ..., x_{n-1}) = \prod_{i=0}^{n-1} p(x_i)$$

$$\text{with } p(x) = \begin{cases} P(x) & \text{if x is True} \\ 1 & \text{otherwise} \end{cases}$$

*Bounding Condition* :

$$Result \geq Probability\ Lower\ Bound$$

**Optimization problem**

The last step of this development is the optimization of the new formula. The goal is to get a minimum value of *Result* for the cost or the maximum value for the probability. This could be done easily by iterating on every solution but it would be very inefficient. Instead, some SMT solvers propose Optimization Engines working by adding objectives with soft constraints to minimize or maximize some variables.

## 2.5 Probability analysis

Another interesting analysis to be done is the risk evaluation of an attack on the root node of the Attack Tree. This risk can be evaluated by finding the success probability to find any successful attack against the root node.

### 2.5.1 Success probability of any attack

The calculation of this success probability is performed under two assumptions : (1) the first one states that every variable represents **independent events** (which might not be true in real situations but will be a very useful approximation for our computation), and (2) the second one states that we have a Tree structure and not a DAG model as **shared nodes will make the procedure obsolete** (as they would need more complex calculations). The computation is performed by a simple bottom-up algorithm which will assign a probability to each intermediate (non-leaf) node by applying the following rules for each type of logic from the leaves to the root until every node has its own probability :

- **NOT** :

$$P_{not} = P(\neg x) = 1 - P(x)$$

- **AND** :

$$P_{and} = P\left(\bigwedge_{i=0}^{n-1} x_i\right) = \prod_{i=0}^{n-1} P(x_i)$$

- **OR** :

$$P_{or} = P\left(\bigvee_{i=0}^{n-1} x_i\right) = 1 - \prod_{i=0}^{n-1} (1 - P(x_i))$$

- **XOR** :

$$P_{xor} = P\left(\bigoplus_{i=0}^{n-1} x_i\right) = p_n = \begin{cases} 0 & \text{if n} = 0 \\ (1 - p_{n-1})\, P(x_{n-1}) + p_{n-1}\, (1 - P(x_{n-1})) & \text{if n} > 0 \end{cases}$$

To achieve this, each leaf node must have its own independent success probability (and if that's not the case it is set by default to 1.0 so it doesn't influence the global result). Note that the generalization of the XOR logic operation uses recursion as the generics mathematical development was too big to be used.

## 2.6 Attack trees as boolean formulas

To solve attack trees with boolean SAT solvers, a boolean formula must be retrieved from the graphical representation. This section will explain the straightforward translation of attack-trees into boolean formula.

**DFS walk-through :** A *Depth First Search* [69] path along the nodes of the tree is able to create a boolean sentence from that tree. Given a tree, the DFS recurrence algorithm explores the structure node by node until a node has no children nodes. Then it backtracks upwards until it finds an unexplored child node. Here is an example of a DFS path on a tree :

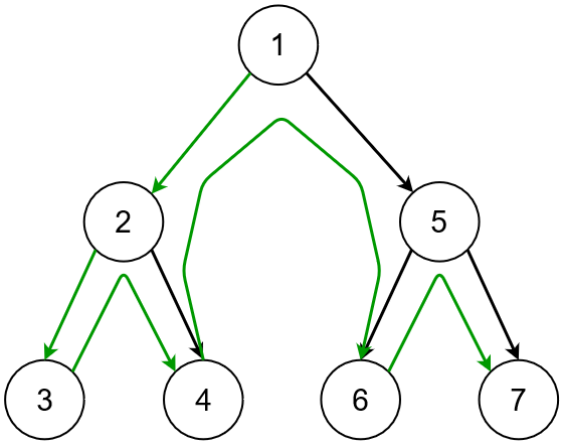



Figure 2.4: DFS walk-through on a tree

Starting from the root node 1, the DFS will explore the node 2 and its children until reaching the node 3 which is a leaf node. Then it backtracks to discover the second child node 4 which is also a leaf node. Another backtracking will detect the node 5, the second child node of the root node. The final path of the DFS along the nodes of the tree is : [1,2,3,4,5,6,7].

Using the DFS walk-through, it is possible to design a function writing the logic boolean formula of a tree. Here is a pseudo-code of this function (algorithm 1) :

**Algorithm 1:** Translation of tree into boolean formula with DFS crossing

```
// node.children → list of children of that node (a leaf node
 has 0 children)
// node.var → boolean variable of the leaf
// node.gate → the logic gate of the node
```

**Data:** Root node of the tree
**Result:** Boolean formula of the tree
**def** *Boolean_translation(n)*:

```
if n.children is empty then
    print(node.var)
else
    print("(")
    for each child of n do
        Boolean_translation(child)
        if child is not the last one then
            print(n.gate)
    print(")")
```

**Example**
This figure represents a small boolean tree to test the presented algorithm.

1. Node1 opens parenthesis "("
2. Node2 opens parenthesis "(("
3. A is a leaf node and prints the variable "((*A*"
4. Node2 add its logic gate "((*A*&"
5. B is a leaf node and prints the variable "((*A*&*B*"
6. Node2 closes the parenthesis "((*A*&*B*)"
7. Node1 add its logic gate "((*A*&*B*)|"
8. C is a leaf node and prints the variable "((*A*&*B*)|*C*"
9. Node1 closes the parenthesis "((*A*&*B*)|*C*)"

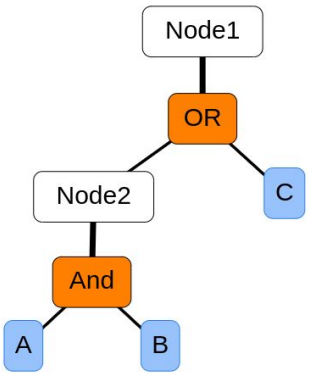


Figure 2.5: Boolean tree

This section showed how to translate a boolean attack tree into a boolean formula. The following sections describe the different analyzes available to perform by solving these Boolean formulas.

### 2.6.1 Attack traces with boolean SAT solvers

Using the tools described in previous sections any attack tree can be translated into logic formula and the list of attacks can be computed with SAT solving computation. This section will define the attack traces of the example in figure 2.5

**First step :** Translate the tree into a boolean formula as explained in previous section 2.6. The translated formula is :

$$( \ ( \ A \ \& \ B \ ) \mid C \ )$$

**Second step :** To be solved, this formula has to be transformed into Conjunctive normal form (CNF) as explained in section 2.3. The transformed CNF formula is :

$$( \ A \mid C \ ) \ \& \ ( \ B \mid C \ )$$

**Third step :** The SAT solver can compute all solutions satisfying that transformed boolean formula. Every solution is a valid list of actions for the attacker to succeed the root node. This table resumes these solutions :

| A | B | C |
|---|---|---|
| True | True | True |
| True | True | False |
| True | False | True |
| False | True | True |
| False | False | True |

Attackers have 5 different combinations to achieve the attack. But watchful readers do see that some of these solutions include other smaller solutions. The set of relevant attacks is the list of attacks which can not be reduced to another valid solution with fewer *True* variables. There are only two solutions left :

| A | B | C |
|---|---|---|
| True | True | False |
| False | False | True |

These two solutions are the two minimal attack traces of the system. This basic example showed how to compute the traces of an attack tree. Note that more complex trees generate a lot more solutions and calculations of solving methods grow up exponentially in the size of the input. That's why using the most effective tools becomes necessary and should not be put aside.

### 2.6.2 Trees comparisons with boolean SAT solvers

Engineers using attack trees to define their systems should try to reduce the number of attacks and their dangerousness against their system. To improve their systems, these engineers will compare different trees to find out which one and which modifications have the best impact on the number of attack paths. A system directly better or equivalent than another system sees all its solutions included in this other system. This system can be considered safer because fewer sequences of actions can break it.

**The comparison of two trees :** The goal is to verify if all solutions of a given tree are also solutions of another tree. In logic, there are several ways to perform this calculation. We assign P symbol to the computed tree and Q symbol to the reference tree. The first possibility is to check if all solutions of P belong to the solutions of Q. The second possibility is to find out if the space of "$P$ & $\neg Q$" is empty. If this space is empty, then P is included inside Q. This second approach involves computing the solutions of the fusion of the trees P and Q which can be intensive.

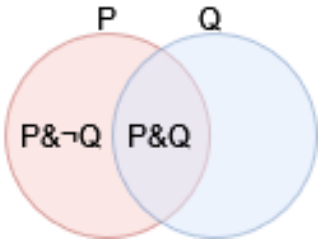



Figure 2.6: Intersection of solution spaces

**Example :** This example will compare two simple trees. The goal is to know if the left tree is included into the right tree. The inclusion works if all the solutions of the left tree are also solutions of the right tree.

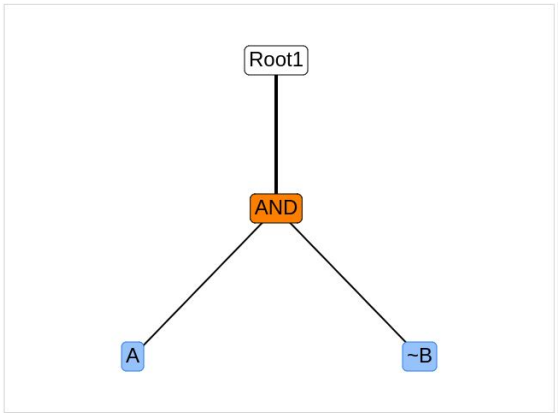



Figure 2.7: Tree P "$(A\&\neg B)$"

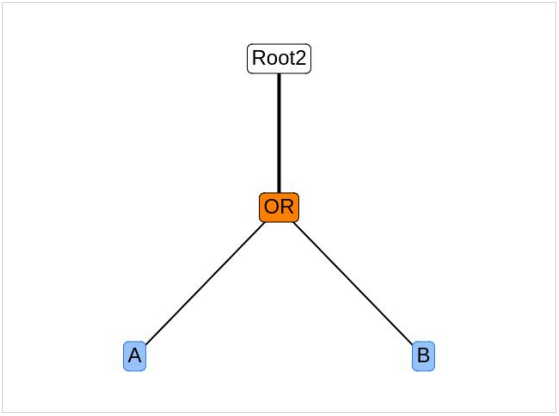



Figure 2.8: Tree Q "$(A|B)$"

The left tree P owns only 1 attack path :

| A | B |
|---|---|
| True | False |

The right tree Q owns 3 attack paths :

| A | B |
|---|---|
| True | True |
| True | False |
| False | True |

As the examples create a small amount of attack traces, it is easy to rapidly spot that every solution of P is also solution of Q. But it is possible to prove it by solving the following formula :

$$( P \,\&\, \neg Q ) \;\Leftrightarrow\; ( A \,\&\, \neg B ) \,\&\, \neg( A \mid B )$$
$$( P \,\&\, \neg Q ) \;\Leftrightarrow\; A \,\&\, \neg B \,\&\, \neg A \,\&\, \neg B$$
$$( P \,\&\, \neg Q ) \;\Leftrightarrow\; A \,\&\, \neg A \,\&\, \neg B$$

As this equation has no solution, the demonstration proves that there exists no solution of P that is not also a solution of Q. The tree P is thus included in the tree Q.

### State of the art conclusion

This section ends the state of the art of this master thesis. The theoretical basis implemented in the tool to successfully display and analyse has been entirely covered. The following sections will present the tool's implementation and features. At last, two use cases will illustrate the possibilities offered by the application.

# Part II

# The tool, model and features

# Chapter 3

# Specifications and methodology

## 3.1 Introduction

This brief chapter will detail the specifications of the project and the methodology behind the theoretical research, the development of the tool and the production of the report. This thesis required a "research" component about the different existing models of attack trees and the diverse used SAT solvers. The “technological development” component is the main part of the project as the goal is the development of a tool.

## 3.2 Specifications

The model of attack tree is already a well-defined model. It is used to represent an attack scenario against a system as well as the defenses deployed on that system. The creation of attack tree becomes tedious with large systems. This project's goals are the followings :

- Propose an attack tree model based on the state of the art and the possible needs of the user.
- Define a grammar to easily define nodes of the tree. The tree should be easily decomposed into multiple sub-tree to use to full advantage of the attack tree model.
- Provide a clear display of the scenarios to take full advantage of the tree structure.
- Allow users to save their trees into files and reload them for reuse.
- Add some analysis features to list the attack paths and compare attack trees of different systems.

## 3.3 Methodology

During the year, the project went through different steps we faced together. Here is the summary of these different steps during the first part of the year :

1. Research around the attack tree literature. Many documents describe features and use cases for feature models and attack trees.

2. Definition of the grammar for the creation of the trees. This grammar had to be simple and offer wide possibilities to the user. Additionally, the tool has securities against misused grammar to avoid unexpected results or crashes.

3. Creation of the tool, implementation of the grammar and the translation between the grammar and the json file and finally the use of a graphical unit to represent the tree.

4. Start of the report with the summary of the grammar and the state of the art.

These were the first steps of development of the application. The second part of the year targeted the boolean quantitative analysis with SAT solvers and the analysis features. Here are the different development steps :

5. Adding a translation from the tree structure to a boolean formula.

6. Addition of a SAT solver to compute the solutions of the boolean formulas of the tree.

7. Implementation of analysis features such as tree comparison and SMT solver.

## 3.4 Implementation

The tool has been mainly developed over *python* and was tested in a Linux environment (Ubuntu OS). The GUI is implemented with the module PyQt5 [68] and nearly all features are implemented with *python* modules. The creation and reading of *json* files have been implemented with *C* functions as explained in section 4.8.

The tool runs a backend behind the frontend creating the different windows. This allows more flexibility in the implementation and the possibility to balance the load of intensive computations among multiple threads while keeping the frontend running in the main thread. The swiftness criteria was a major concern as we wanted to avoid the tool to freeze during intensive computations. Here is the list of swiftness optimisations implemented into the tool :

- The creation and reading of *json* files are done by specialized functions from *C* classes. These compiled functions are faster than *python's* interpreted code.
- Worker Classes have been defined to split up critical computations with multi-threading procedures. The critical computation term includes (1) the grammar parsing, (2) the solving with SAT solver and (3) the resolution of the SMT equations.

## 3.5 Introducing the tool

The github repository of the tool is available at the following url : `https://github.com/weytho/AttackTree`. The following sections explain the installation procedure and briefly familiarize you with the interface of the main window.

### 3.5.1 Installation and requirements

An installer script is provided within the tool's implementation (*install.sh* file). It installs the required Python libraries which are listed in the *requirements.txt* file and a few other necessary dependencies (to manipulate json file in C or packages which can't be installed with the pip command). Then the script sets the working directory (its location directory) and installs the implementation as a package in the environment thanks to the *setup.py* file. The package is called AttackTree_MAGI for "**Attack Tree Modelling and Analysis Graphical Interface**" (shortened in at_magi). This file also compiles the C extension used in the project. Once installed the tool can be called from any terminal with the command : "atmagi_launcher". The global requirements are as follows :

- json-c [5]
- Python 3.8
- graphviz
- numpy
- cython
- pandas
- matplotlib 3.4.3
- networkx 2.6.3 [35]
- PyQt5 5.15.6 [68]
- PyQtWebEngine
- pysat 3.0.1 [38]
- python-sat
- PySMT 0.9.0 [32]
- pyvis 0.1.9 [6]
- graphviz
- pygraphviz
- setuptools 52.0.0
- sympy 1.9 [60]
- z3-solver 4.8.17.0 [8]

### 3.5.2 Main interface description

Once installed, the application opens on the main window. In this window, 4 main components are distinguishable :

1. A text window to write down the tree using the grammar defined in chapter 4. Buttons allow the user to translate this written version of the tree into the *json* file version. It is possible to define any name for that file.

2. A display window for the graphical representation of the tree. To display a tree, import the tree's *json* file the application. An interactive tree will be displayed in the middle of that window.

3. A feature column proposing various options : An output format for the boolean formula, the possibility to display precise solutions on the tree, a generator of random tree, an option to fix the value of leaves and countermeasures, a tool to verify if a tree is included in another, a SMT solver,...

4. A solution screen to show the boolean formulas or the highlighted solutions.

See the appendix A for a complete overview of the tool and the description of each screens and options.

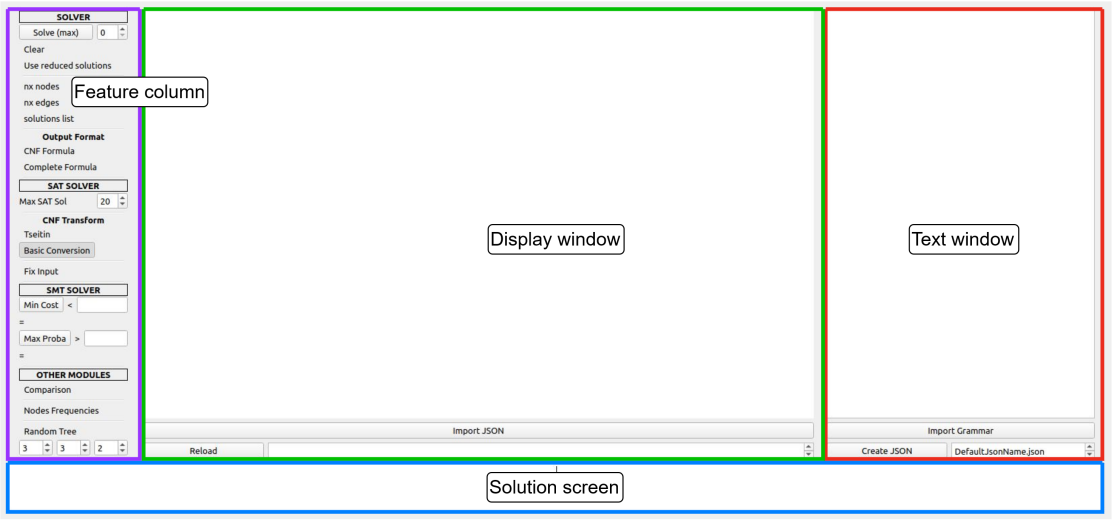



Figure 3.1: Main screen of the tool

# Chapter 4

# The proposed tree model

## 4.1 Introduction

This section provides a full description of the application and all the needed information to correctly create and understand attack trees. This chapter explains the deployed grammar to describe the trees in an understandable way for the application. This grammar has been designed to be simple and effective. As described in section 1.3.2 the attack tree model used in the tool is a DAG-based structure with a few basic gates and two numerical properties for quantitative analysis. This model has a high improvement potential as new gates and metrics can enhance its expressiveness.

## 4.2 Grammar overview and tree example

The created grammar is in charge of the construction of the tree according to the tree model described in section 1.3.2. This grammar allows (1) to design edges between parent and child nodes, (2) to give a meaning to these edges with logic gates, (3) to define countermeasures on the system's nodes and finally (4) to add numerical properties to the leaf nodes. This section explains the structure of this grammar and illustrates it with a simple example. A first view of the 3 paragraphs segmenting the grammar is as follows :

**RELATIONS** keyword opens the first mandatory paragraph describing the links between the nodes and edges of the tree. Each line of the paragraph mentions a parent node, a logic gate and a list of child nodes linked to this parent throught the logic gate. See section 4.3 for a deeper view of this paragraph.

**COUNTERMEASURES** keyword opens the optional paragraph describing the different defenses protecting the system's nodes or weakening the attacker's action nodes. Each line mentions the name of a countermeasure and the list of nodes it is linked to. See section 4.4 for a deeper view of this paragraph.

**PROPERTIES** keyword opens the optional paragraph giving properties to nodes and countermeasures. Each line mentions the name of a leaf node (not an intermediate node) or a countermeasure and binds it to a cost and a success probability. 4.5 for a deeper view of this paragraph.

The following frame lists the defined syntax to declare the tree grammar paragraphs and their grammar lines :

```
RELATIONS
node -logic_gate- list_of_children
COUNTERMEASURES
cm_name (list_of_nodes)
PROPERTIES
leaf_name:cost= COST, prob = PROB
```

To illustrate this grammar, the following frame contains a small example illustrating nearly all different possibilities :

```
RELATIONS
NODE -OR- A,B;
A -SAND- C,D;
B -AND- D,E,F;
COUNTERMEASURES
CM1 (E)
PROPERTIES
C:cost = 1, prob = 1.0;
D:cost = 1, prob = 1.0;
E:cost = 1, prob = 1.0;
F:cost = 1, prob = 1.0;
```

This example defines a valid tree with a root node (NODE), 2 logic nodes (A, B) and 4 leaf nodes (C, D, E, F). A countermeasure (CM1) is linked to the node E meaning that the defender can alter the cost and/or the success probability of that node E. Finally, all leaves of the tree have the same cost and success probabilities.

Once the tree is written in the text window, generate the *.json* file by clicking on the **Create JSON** button. Then use the **Import JSON** button and select the newly created file. The tree should appear on the screen and look like figure 4.1. The following sections go deeper in the explanation of the grammar.

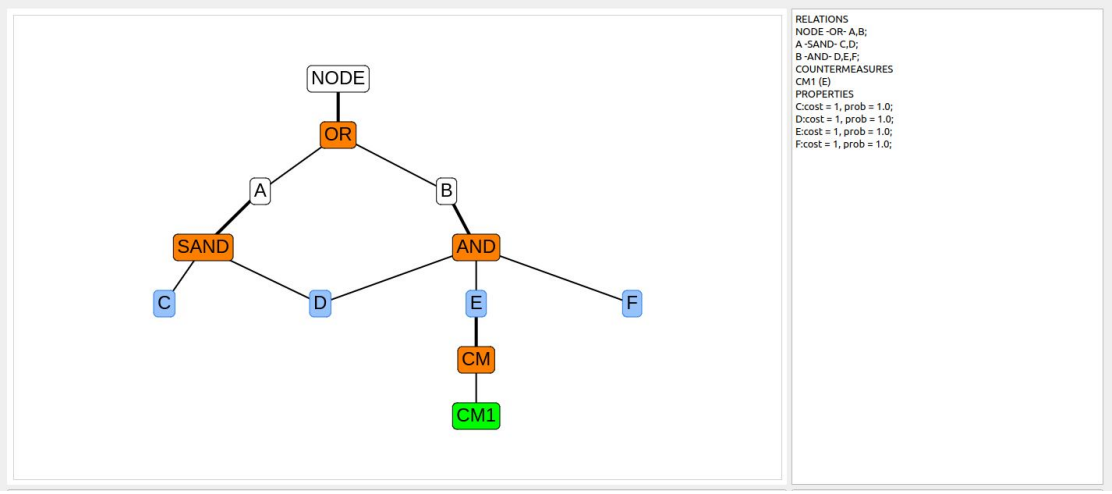



Figure 4.1: Simple tree created with the grammar example

## 4.3 Relations

The paragraph starts with the keyword "RELATIONS" and will enumerate the relations between 1 parent node and several child nodes. No mandatory order is required by the parser to understand the given grammar lines. Five different logic gates are understood by the SAT solver : OR gate, XOR gate, AND gate, SAND gate and NOT gate.

- OR gate : If at least one of the child node is satisfied then the parent node is satisfied.
- XOR gate : An odd number of children of the node must be satisfied to satisfy the parent node.
- AND gate : All children of the node must be satisfied to satisfy the parent node.
- SAND gate : The Sequential AND gate also needs all its children to be satisfied but requires those children to be resolved in the given order. For example, an attacker must first get inside the system before harming it. Note that this gate behaves like an AND gate for boolean solving as the tool does not use a temporality metric.
- NOT gate : This special gate only works for parent node with 1 child node. The parent node is satisfied if the child node is not.

Note that any string name can be used to define the name of the logic gate but only the listed gates will be understood by the SAT solver. This can be useful to express other non implemented gates in graphical representations.

As illustrated by the previous example, the nodes can have **shared children** with other nodes. This increases the expressiveness of the model as, in attack scenarios, an action can be a mandatory step for multiple higher-level attack steps. The possibility to group multiple repetitions of the same action into a single node is a visual benefit to display the tree but also an information gain for the attack path analysis.

## 4.4 Countermeasures

This paragraph allows the creation of countermeasures as described in the model. Note that it is not mandatory to define countermeasures in an attack tree. This paragraph starts with the "COUNTERMEASURE" keyword and the grammar lines bind a countermeasure node with a list of existing action nodes. It is currently not possible to assign metrics to the countermeasures. See the areas of improvement section 7.1 for a brief discussion about this option.

## 4.5 Properties

The last paragraph starts with the "PROPERTIES" keyword and permits to bind two numerical parameters to the leaves of the tree : the **cost** and the **success probability**. The costs represent the "difficulty" to succeed the step of the attack and the probabilities represent the success rate of the actions. Note that the tool implicitly sets a cost of 0 and a success probability of 1.0 to any leave without properties.

## 4.6 Implementation

Once the text of the grammar is sent to the parser function, a few verifications are made to separate the three categories, find errors in the grammar definition and progressively build the tree. If there is no Properties the parser is restricted to boolean tree structure. A relation should always be of the form of :

$$\text{Parent - Operator - } Child_1, Child_2, \ldots ;$$

To parse the Relations and create the tree structure we have to process each relation separately by using the end of relation sentence delimiter ";". By cutting the obtained relation on the "-" and "," delimiters we can get all the elements of a relation and create a node structure to create the tree. The implemented tree architecture is showed in figure 4.2. The node and tree structures which were used are :

```
struct Node
    char title[50] // node name
    char variable[50]
    char type[5] // type of logic
    int root
    int leaf
    int cost
    double prob
    int CM
```

```
struct Tree_LL
    Node* n // current node
    Tree_LL *next // siblings node list
    Tree_LL* parents // parents nodes list
    Tree_LL* children // children nodes list
```

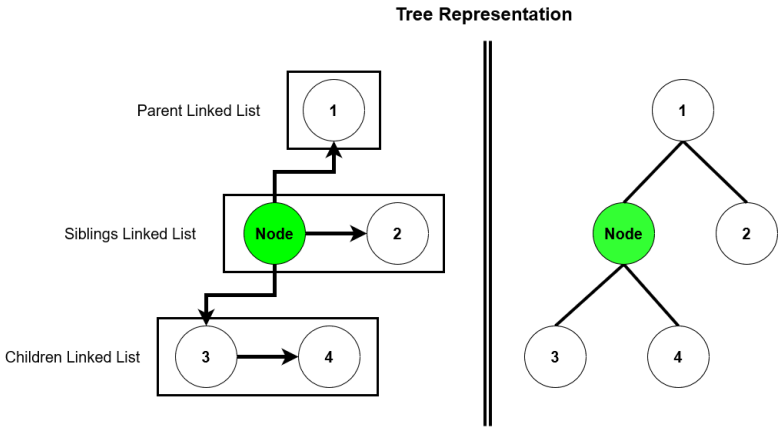



Figure 4.2: Tree using Tree_LL structures

The errors returned when creating the tree are shown in a pop-up window and can be of 5 different types :

- Error(1) = **Nothing to parse** : when the given text is empty.
- Error(2) = **Rewrite existing relation** : when the parent of a relation already has children assigned.
- Error(3) = **Relation creates a cycle in the tree** : when a relation has the parent among the children or when its addition to the tree creates a cycle (this is verified by recursively iterating from child to parent(s) until either root is reached and there is no cycle or we find back the original child, which means that a cycle was created).
- Error(4) = **Multiple roots are present** : when the root has siblings nodes after every relation was added.
- Error(5) = **Unary operator with multiple children** : when a negation relation tries to add more than one child.

The complete parsing process is described in algorithm 2.

**Algorithm 2:** Parsing and Creation of the Tree Structure

```
Data: RelationsText (RelText)
Result: Attack Tree Structure
if RelText is NULL or empty then return Error(1)
relation ← get first Relation from RelText
tree = NULL // init tree structure
while relation != NULL do
    if relation is not empty then
        nodename ← get Parent name from relation
        if nodename != NULL then
            type ← get Type from relation
            if nodename not in tree then
                parent node ← create node and add it to the roots list
            else
                parent node ← get existing node "nodename" from tree
                if Parent already has Children then return Error(2)
                parent node→type = type
            nodename ← get Child name from relation
            while nodename != NULL do
                if nodename is not empty then
                    if type is NOT and children count > 1 then return Error(5)
                    if nodename[0] == '∼' then
                        create negation relation to parse at the end : "∼name-NOT-name;"
                    if nodename not in tree then
                        child node ← create a new leaf node called nodename
                    else
                        child node ← get existing node "nodename" from tree
                        if Relation creates cycle then return Error(3)
                        if Child already has Parent then
                            child node ← extract existing node without its siblings pointer
                        else
                            child node ← copy existing node without its siblings pointer
                    add Child to the Parent Node
                    add Parent to the Child node
                nodename ← get next Child name from relation
    relation ← get next Relation from RelText
    if relation == NULL then relation ← all the implicit negation relations to parse
if Root node has next node then return Error(4)
add properties, countermeasures and create Json file
return 0
```

## 4.7 Random Tree Generator

This section describes the integrated feature generating random trees based on the described grammar. These are useful to do general tests on the analysis features and confirm that the graphical representation deals well with cranky trees. This feature is implemented in python and requires 3 arguments to orient the random generation.

**Argument 1 : maxdepth.** This numerical argument prevents the random tree to exceed a certain depth. Note that it is possible that the generated tree has a smaller depth than the value of this argument. Increasing this number will increase the length of the tree.

**Argument 2 : branching_factor.** This numerical argument affects the maximal number of children that can be generated under a single parent node. Increasing this number will increase the width of the tree.

**Argument 3 : complexity.** This numerical argument defines the complexity of the tree. The following list enumerates all the different tree generators :

- Complexity 1 : This complexity will generate the most simple tree with relations, countermeasures and properties.
- Complexity 2 : This second level of complexity adds shared children and shared CM to the level 1 of complexity.
- Complexity 3 : The third level adds negative leaves, NOT gates and XOR gates to the level 2 of complexity.
- Complexity 4 : The fourth level of complexity adds more randomness on the generation of shared children of the second level of complexity. the generated tree might be inconsistent.
- Complexity 5 : This last level of complexity shuffles the lines of the grammar of the level 4 of complexity.

Here is the pseudo-code of the tree generator (algorithm 3). The generation of countermeasures and list of children are random but are capped to the branching_factor argument. **Important note :** The raise of the *maxdepth* and *branching_factor* arguments induces an exponential growth in the number of generated nodes. Around a maxdepth and a branching_factor of 10, the random calculation slows down for the creation of certain trees. It naturally gets worse for higher sized trees.

**Algorithm 3:** Generation of random tree with the grammar

**Data:** maxdepth, branching_factor, complexity
**Result:** A random tree accorded to the input variables

```
// Main function selecting the right complexity to call
def TreeGen(maxdepth, branching_factor, complexity):
    R = "RELATIONS"
    C = "COUNTERMEASURE"
    P = "PROPERTIES"
     // Setup right complexity
    R,C,P = nodeGeneration(R,C,P,"node",0,maxdepth,branching_factor)
// Recursive function generating a random text describing a tree
def nodeGeneration(R,C,P,node,0,maxdepth,branching_factor):
    if randomstop() then
        // This node is a leaf
        addProperties(P,node)
        return R,C,P
    addNode(R,node)
    addLogic(R,node)
    addCM(C,node)
    child = generateChildrenList(node,branching_factor)
    if depth==maxdepth then
        // These nodes are leaves
        addProperties(P,child)
        return R,C,P
    for e in child do
        // Following generation for the children
        R,C,P = nodeGeneration(R,C,P,e,depth+1,maxdepth,branching_factor)
    return R,C,P
```

## 4.8 The Json file structure

The tree created with this grammar can be transformed and saved into *.json* file form. These saved files will allow the users to keep versions of their previous trees, display them again and use them to do comparisons to improve their systems. This section describes the structure components of this file.

### 4.8.1 Structure and fields of Json file

Once the tree has been written with the defined grammar, the **create json** button will translate it into a *.json* file. The file enumerates all nodes of the tree starting from the root node. Three different types of nodes are used inside the structure :

*Parent node* : The **Type** attribute of the node is a logic gate and the **Child** attribute lists all the nodes behind this node. The **Action** attribute of this node will be the displayed name of this node on the graphical tool.

*Leaf node* : The **Type** attribute of the node is **LEAF** and has no **Child** list. The **Action** attribute of this node will be the displayed name of this node on the graphical tool and will be the name of the boolean variable used to solve the tree. Two additional arguments can be used to specify properties like **Cost** and **Proba**. These are used by the SMT solver to compute the total cost and probability of the attack paths. If no properties are specified, all leaves are considered to have a cost of 0 and a probability of 1.

*Countermeasure node* : This kind of node is different from the two nodes mentioned above. It represents the actions of the defenders of the system. The goal of these actions is to nullify the actions of the attacker or at least reduce their success probabilities. The **CMtitle** attribute is the name of the node displayed by the tool. The two optional attributes **CMcost** and **CMprob** represent the numerical metrics associated to the countermeasure. These countermeasure metrics are merely present in the structure as an example as they are not yet used by the analysis tools.

An example for this tree structure can be found in the annex C. This file corresponds to the example of chapter 4.

### 4.8.2 Implementation

The creation and reading of these *json* files are implemented with *C* functions for speed reasons. These operations need to be fast as huge trees require more backtracking and more computation. Specialized *C* functions are more effective for

this kind of operation. Basically, these functions recursively explore the different branches of the tree and translate them into either a *json* file, either a format used by the solvers.

### Creation of a *json* file

With the *libjson-c-dev*[5] library it is possible to convert our tree structure to a JSON file. This is done by creating json objects and adding their respective fields with the appropriate data, using json arrays for children or countermeasures (by using the hashtable) when going down in the tree. To traverse the tree we just have to recursively call the function on the children of a node until we get to the leaves. Then the complete json object is written to the output json file with the specified filename.

### Reading from a *json* file

Using the same library, the tool can read the different json file attributes and encode it into lists of tree structure in *C*. While reading the file recursively node per node, the function fills up 2 lists of objects :

- A node list collecting all nodes and their attributes.
- An edge list creating all links between these nodes.

These structures are translated to the main tool implemented in *python* using python *ctypes* [1] library. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

# Chapter 5

# Built-in analysis features

## 5.1 Introduction

As discussed in the state of the art, attack tree models offer a pleasant display of complex attack scenarios but are also used to assess them. This chapter resumes every feature available in the application. Certain come from the attack tree literature while others are more related to the grammar of the tool. Some features will only be briefly mentioned as they have already been developed in previous chapters and no supplementary remarks are needed.

## 5.2 Ease the use of the grammar

One of the tool's requirements was to propose an expressive but simple grammar. The number of mandatory elements to create a basic tree has been minimized as much as possible and some features have been developed to assist users in the creation of their trees :

- The first feature is implemented directly in the parser and looks for mistakes in the input grammar. The tool notifies users whenever they wrote an erroneous tree.
- The second feature allows users to import data from a *txt* file apart from the *json* file. This enables users to save and reload previous grammar trees to add modifications.
- The last feature is the random tree generator described in the section 4.7. This feature was used to assess the performances of the tool but is also an assistance to users generating tremendous trees.

## 5.3 Display window

The python libraries displaying the tree are Networkx and Pyvis. The first one creates a DiGraph representation of the tree and assigns hierarchical positions thanks to graphviz extension. The second one uses the Networkx information combined with some parameters loaded from a json configuration file (pyvis_param.json) to create a network representation in html which is then rendered in a QWebEngine. Additional information can be shown when hovering a node and we can interact directly with the graph (moving nodes, zooming and reloading initial html). These libraries are in charge of the creation of the tree once all nodes and relations are declared. This section will describe the meaning of the different nodes displayed in the window. The names and properties of the nodes of the trees are displayed during mouseover.

- **Intermediary node** : These nodes are white and represent the intermediate steps of the attack. These intermediate steps are always matched with a **logic gate node** colored in orange. This logic gate is linked to the children node of the logic node.
- **Leaf node** : These nodes are blue and are the nodes at the end of any branches of the tree. They are indivisible actions and serve as variable name to find the solutions of the tree.
- **Countermeasure node** : These optional nodes are green and represent the defensive actions added to the system. For the moment, they have no impact on the computation of the solution except that they can paralyse an action and force it to be false.

## 5.4 Solving the Attack Tree

In order to get the set of variable assignments which are able to satisfy the boolean expression of the tree and find their properties in term of cost and probability a SAT solver and a SMT solver are used. To avoid freezing GUI during calculation, the implementation uses QThread to create Worker threads which synchronize with the main thread on completion.

### 5.4.1 Attack Satisfiability with SAT solver

The reasoning behind SAT solver implementation was previously stated in section SAT solvers. We used SymPy library to create the formula by parsing our string expression of logical operations. Then, for our implementation we used two different

CNF transformation. The user has the choice between basic conversion (with sympy built-in method) and Tseitin transformation. We built our own implementation for the latter which is described in the annex 5. Once a CNF form is obtained, it is sent to a SAT solver. The SAT solver we used was implemented thanks to PySAT library which is a Python interface that regroups many solvers from faster low-level languages. Among those SAT solvers we choose **Glucose 3.0** ([12], [13]) which is a Minisat-based solver. As such it is using CDCL algorithm and combines it with aggressive restarts and heuristics which sacrifice completeness to enhance their good clauses identification and bad clauses deletion. The solver will return the list of variables with their truth values for the number of solutions required by the user. To get more than one solution the solver has to block each previously computed model by adding a new clause to the formula (negation of the solution found) until we have enough solutions or no more models exist.

### 5.4.2 Attack Cost and probability with SMT solver

Like for SAT problem, SMT solver were previously described (in section SMT solvers). This time we use the pySMT library ([32]) which provides an interface to manipulate formulas and call multiple SMT solvers API. Among the supported solvers the **Z3Prover** (or just Z3, with related implementation papers here [61], [19]) seemed like a good choice as it supports most theories, is quite efficient and has a lot of additional features. The Z3 solver is especially useful in our case as it provides an optimization module for our optimization problem where we want to minimize a cost or maximize a probability. We can easily use the minimize and maximize functions on the result of the evaluation function to get a solution. In term of theories it is a bit more complex. We can define the two used theories as :

- QF_LIA logic for cost computation
- QF_NRA logic for probability computation

In this notation QF means Quantifier Free (no quantifier of any sort in the formula), L is for linear arithmetics (sum of variables in evaluation function), N for non-linear arithmetics (multiplication of variables in evaluation function), IA for Integer arithmetic (costs are Integer) and RA for Real arithmetic (decimal probabilities are Reals). Remarkably, even if the documentation stated that the optimization for non-linear arithmetic isn't supported (simplistic non tested approach is said to be used instead) by Z3, it is able to find quickly the SMT solutions for the probability computation. It might be due to the fact that we use boolean variables which can only be assigned 2 values for its probability (its own or 1.0) and which could enable simplification or other underlying techniques.

### 5.4.3 General probability of attack success

Using the equations from section Success probability of any attack the probability that any attack enables a node is calculated by the following algorithm (4). However the calculation takes place exclusively when there is no shared node inside the Attack Tree. To verify that the analyzed structure is a Tree and not just a DAG we counted the number of time each node is used as a child (by using the edges of the graph) and if the count of a node exceed 1 then it means that it has more than one parent. If the structure is a tree we add the probability of each intermediary node in their description (showed when hovering the node in the plot window).

**Algorithm 4:** Success probability calculation

```
Data: List of nodes with their properties (ln) and List of edges (le)
Result: Python Dictionary of nodes with their respective probability
// Compute Global Probability of Successful Attack
def get_global_proba(ln, le):
    leaves ← list of nodes names with type 'LEAF'
    dict_proba ← dictionary of the initial given probability of each node
    for every node in leaves do
        while current node is not None do
            if current node is a leaf then pass
            else if n logic is 'NOT' then
                dict_proba[current node] = 1 - dict_proba[child]
            else if n logic is 'OR' then
                proba = 1
                for each child of current node do
                    proba = proba * (1 - dict_proba[child])
                dict_proba[current node] = 1 - proba
            else if n logic is 'XOR' then
                proba = 0
                for each child of current node do
                    proba = (1 - proba)*dict_proba[child] + (1 - dict_proba[child])*proba
                dict_proba[current node] = proba
            else
                proba = 1
                for each child of current node do
                    proba = proba * dict_proba[child]
                dict_proba[current node] = proba
            current node ← get parent node from edges or assign None
    return dict_proba
```

## 5.5 Tree solution path coloring

Once we get a solution we can show it by plotting it on the tree. To do this we color in red the leaves nodes which were assigned to True and by travelling upwards in the tree successively while checking the logic (AND are colored when all children are colored, OR when at least one, ...) we can color the rest of the activated nodes. The trace created that way represents all the steps used in the attack to access the root. When coloring the nodes a backup configuration is also saved to be able to return to the default state where no solution is highlighted.

## 5.6 Reduction of solution space

This feature has been added in response to the cumbersome number of redundant solutions generated by the OR gates. In logic, the OR gate is satisfied if at least one input of the gate is true. In the context of attack tree, this behavior is not intended. Indeed if multiple actions achieve the same goal, the attacker would only perform one of these actions to succeed that goal. An attack path performing two actions of an OR gate is obviously not optimal because it could achieve the same goal with just one action. These attack paths are qualified as redundant and exponentially increase the number of solutions of the an attack tree. This is an inconvenient in two ways : (1) the SAT and SMT solvers computes a high number of redundant solutions and (2) the user is drowned into a high number of solutions.

A possibility to solve this problem is to create the *"1 out of N"* gate. This gate behave like an OR gate but is not satisfied if more than one input is true giving the following truth table :

| OR gate | | |
|---|---|---|
| A | B | A or B |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

| 1 out of N gate (1n) | | |
|---|---|---|
| A | B | A 1n B |
| True | True | False |
| True | False | True |
| False | True | True |
| False | False | False |

Unfortunately, this gate has not been implemented in this tool and this topic is discussed in section 7.1.

Another solution to tackle the problem has been implemented in this tool. The idea is to reduce the computed solution space to only non redundant solutions. Basically, this feature loops over the list of solutions and sorts only the non redundant solutions. The output is exactly the same as the output of the *"1 out of N"* logic gate but this only solves user's problem by reducing the number of

displayed solutions. Indeed, the SAT solver still have to compute an tremendous number of solutions due to the OR gate which is not optimal. Moreover, the reduction itself is an additional computation done over a potentially large number of solutions. This upgrade is another area of improvement of this tool.

## 5.7 Fixation of boolean variables and countermeasures

A way to fix specific nodes was also added for satisfiability solving. With a simple array of buttons corresponding to every node (countermeasures included) we can force specific nodes to be activated or deactivated wehn computing the SAT problem of the tree (see figure A.5). This is done by adding assumptions to the formula used by the solver and therefore by restricting the solutions set. When using countermeasures the formula becomes a bit different as they don't appear in the default formula. This is why, when the boolean formula is created we make a second one with the countermeasures by defining the simple following constraint (for each variable with at least one countermeasure) :

Node $X$ with countermeasures $CM1$ and $CM2$ becomes $(X \wedge \neg CM1 \wedge \neg CM2)$

- *example* : $X \wedge (A \vee B)$ becomes $(X \wedge \neg CM1 \wedge \neg CM2) \wedge (A \vee B)$

Replacing the variables like this enables the correct computation of the assumptions when countermeasures are present. The color code used for the buttons is green for activated, red for deactivated and unchecked for "don't care". Using this is very useful to see the impact of the countermeasures or the importance of specific nodes on the global feasibility of the Attack Tree.

## 5.8 Tree comparison

This feature is already deeply defined in section 2.6.2. The goal is to allow the user to compare system trees for classification and ranking. This section will describe the specific system used to tackel this challenge :

**Input :** Two *json* files representing the trees. The first one is the tested tree. The tool will compute if its solution space is included in the solution space of the second tree. Here is how the link between the variables of the two trees is handled :

- Any boolean variable repeated between the two trees will be considered as the same variable.

- Any boolean variable present in one tree but not present in the other will extend the solution space of the second one. But the satisfiability of this extended tree will not depend on this additional variable.

**Output :** The output of this feature lists out all solutions of the first tree that are included into the second tree and separately the list of all solutions of the first tree that are not included into the second tree.

**Example :** The following example compares two simple trees and illustrate the principle of extension of the solution space of a system. The first tree to compare on the left has a unique solution and uses 2 boolean variables. But as the second tree uses an additional boolean variable **C**, its solution space is extended with this new variable, doubling the number of solutions :

| A | B | C |
|---|---|---|
| True | False | True |
| True | False | False |

The second tree to compare on the right has 3 solutions and uses 3 boolean variables :

| A | B | C |
|---|---|---|
| True | True | False |
| True | False | True |
| True | True | True |

It is possible now to compare the solutions of the two trees and conclude that the solutions of the first tree are not entirely included in the second tree. Indeed, the second solution is not a solution belonging to the solution space of the second tree.

| Included | | |
|---|---|---|
| A | B | C |
| True | False | True |

| Not included | | |
|---|---|---|
| A | B | C |
| True | False | False |

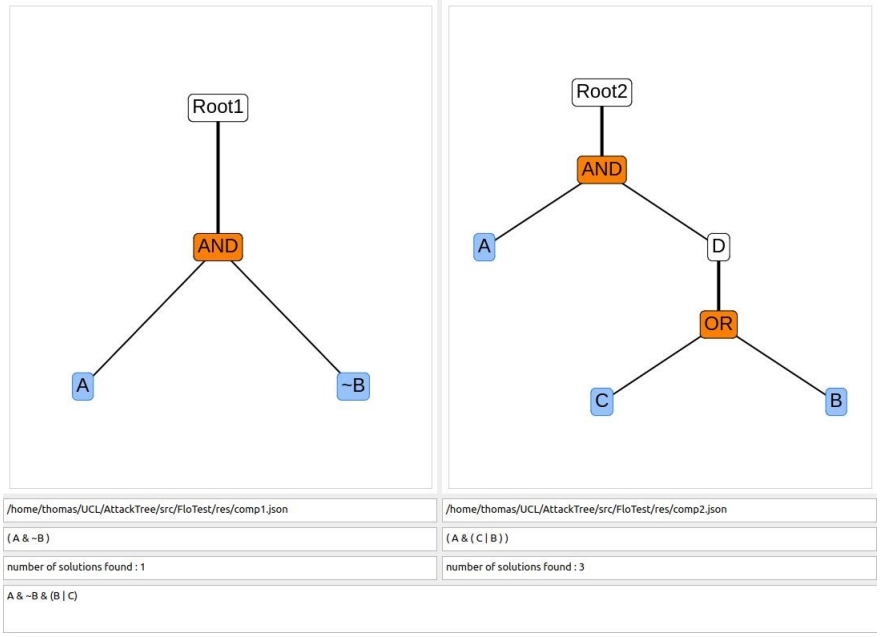



Figure 5.1: Example: Comparison of 2 systems

## 5.9 Nodes frequencies

Another added feature is a visualization of the use frequencies of each node in all the currently computed solutions (see figure A.7). It is displayed of two parts, the left one plots the direct tree of the relations between the nodes and increase each node size according to the number of time they were used in a solution (the color is also set to red for a node selected more than 80% of the time, orange for less than 80% but more than 40% and lime for less than 40%). The second one is a bar chart to plot the frequencies found for each node. To compute these values, the function has to iterate over the solutions and traverse the tree from leaves to root while assigning respective truth value to each node. The main goal of this feature is to visualize which parts of the tree are absolutely needed to realize an attack and which branches or nodes could be ignored in most cases. This permits the user to see the most effective countermeasures to add to the graph.

# Chapter 6

# Case application

## 6.1 Introduction

This section illustrates the tool's ability to display concrete attack scenarios with a high readability. The first use case concerns the modeling of Mirai malware's infection steps and the possible countermeasures against it. The second use case concerns a general DoS attack that could be performed by a botnet like Mirai.

## 6.2 The attack tree of the Mirai malware

This section is a tool use case based on the infamous Mirai [10] malware infecting Linux IoT devices to turn them into a botnet, a large amount of compromised devices listening to a master. Mirai botnet was first spotted in August 2016 and has been used for multiple high scale DDoS attack [79].

This use case will build the attack tree of the Mirai malware. The goal of the attack is to penetrate a new device and add it to the botnet. This list resumes the attack steps of the Mirai malware to infect a new IoT device and integrate it into the botnet :

1. Scan internet for IP address of Linux Internet of things (IoT) devices running telnet, SSH, HTTPS, FTP, or CWMP [4]. Victim IoT devices are identified by stateless TCP SYN probes to pseudo-random IPv4 addresses, excluding some in a hard-coded IP blacklist.

2. If an IoT device responds to the probe, the attack then enters into a brute-force login phase. During this phase, the attacker tries to establish a telnet connection using a predetermined list of 60 common factory default or generic pairs of usernames and passwords.

3. If the IoT device gives an access, the victim's IP, along with the successfully used credential is sent to a collection server.

4. Upon infection Mirai will identify any "competing" malware, remove it from memory, and block remote administration ports to hide from other possible scans.

5. Devices infected by Mirai continuously scan the internet for the IP address of Internet of things (IoT) devices and monitor a command and control server which indicates the target of the future DDoS attacks.

This list resumes high-level countermeasures against the spreading of Mirai and also precises countermeasures that can be applied to the IoT devices to increase their robustness against Mirai :

- Use stronger credentials than the default one. The Mirai brute force is unable to penetrate well designed passwords.

- Apply default networking configurations limiting remote access to those devices to local networks or specific providers.

- IoT designs should apply isolation boundaries and principles of least privilege to prevent connected malware to perform unauthorized actions.

- Install a notification fallback system to request update or infection elimination [54].

- Use honeypots or network telescopes to spot Mirai's scanning phases and characterize its attack.

- A device remains infected until it is rebooted, which may involve simply turning the device off and after a short wait turning it back on. But unless the password is changed immediately, the device will be reinfected within minutes.

The last upgrade to be added to the system are the numerical metrics. The model allows the use of the cost and success probability metrics. The following list describes how we decided to evaluate the attributes :

- Cost attribute : the attack steps of the Mirai infection are extremely low cost as the intrusion only involves TCP connections towards random IPv4 addresses and the infection is performed once connected to the device.

- Success probability : conversely the attack steps are less likely to succeed but this is compensated by a higher number of attempts on many IoT devices.

**Results**

The results of this use case are available in appendix B.1. Screenshots of the tool show the attack's translation into grammar and graphical representation.

This use case illustrates the ability of the tool to achieve the first attack tree model key idea by providing a clear and readable representation of this attack scenario. An uninformed reader is able to understand the related attack thanks to the visual tree.

The tool also computes the different attack paths of the system. The following figure displays the 10 non reducible solutions of the system. Note that the total number of solutions including the redundant solutions is 93 due to the OR gates.

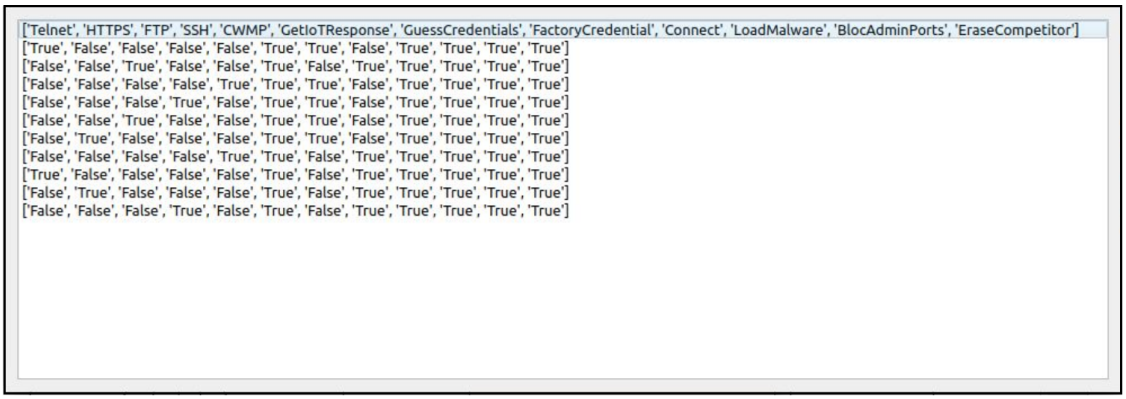



Figure 6.1: Reduced solutions of Mirai attack tree

The Mirai malware has been illustrated with our tool and this model can still be improved. The actions of Mirai's botnet were as spectacular as they were simple. More subtle malware have been elaborated based on Mirai's source code [45]. For example, it would be an easy task to add the behavior of the Mirai-based malware Reaper [7] to the tree model.

## 6.3 Denial of Service attack

This section is about a second use case of the tool and will model a Denial of Service attack (DoS) scenario. This kind of attack targets a device or a server to prevent it to serve the legitimate requests from its client due to a massive data flooding or resources exhausting. The different ways to perform such attacks will be resumed on this use case. The continuity with the previous use case is that the duty of large botnets like Mirai is to perform high-scale Distributed Denial of Service attacks (DDoS).

Many different DoS attacks have been designed and some behavioral rules allow researchers to classify them [3] :

- Volume-based attacks are measured in bits per second. Their goal is to exhaust the victim's bandwidth with massive data flooding. These attacks are generally based on a large botnet and don't go unobserved as they can paralyse devices and servers during hours. Such botnets can be built with worms of malware like Mirai. Here is a non exhaustive list of volume-based DDoS attacks :
  - UDP flood : floods a target with User Datagram Protocol (UDP) packets.
  - ICMP flood : overwhelms a target with ICMP Echo Request (ping) packets. [79].
  - Distributed Reflected Dos (DRDoS) : sends requests with the spoofed address of the victim as source to send back massive amount of valid responses.
  - Amplified DRDoS : is based on the same technique but uses legitimate robust intermediate services (like DNS) replying tremendous amount of valid data to the victim [70].
- Protocol attacks attacks are measured in packets per seconds. They consume the target server's resources with forged packets and connection saturation. These sneaky techniques are still disruptive but are also less likely to be spotted. Here is a non exhaustive list of protocol DoS attacks :
  - SYN flood : spawns half-open TCP connections with SYN packets until saturation [28].
  - Ping of death : manipulates maximum segment size of ping requests to cause an integer overflow in the Linux kernel [81].
  - Slowloris : keeps HTTP-POST TCP connections open for very long time by using extremely low transmission rates and very large content-length header [23].
  - Perpetual Echo : inserts looping UDP echo requests going from the spoofed victim's address to the victim's UDP echo port indefinitely [30].
- Application Layer attacks (Requests per second) are very specialized and aims at crashing a web server using a breach. Seemingly legitimate, these malicious requests are forged against a precise web service.
  - HTTP Flood : forces the server or application to allocate the maximum resources possible in response to every single request.

  - Sophisticated DDoS Attacks : aims at a weak point in the victim's system design and might pass through flow control mechanisms [16].

- Zero-Day Dos attacks are unknown attack paths. More generally, the hacking term "Zero-Day" represents all the exploits based on undisclosed or currently non-addressed vulnerabilities [17].

Against this large variety of DoS attacks, some countermeasures have been deployed to increase the resilience of victim's devices and servers :

- Upstream filtering : sends victim's traffic to pass by a cleaning center removing bad traffic.

- Blackholing and sinkholing : redirects attacked IP address or entire DNS server in a black hole to lower victim device's overload. These techniques have to be used wisely to avoid supplementary negative impacts on the network.

- DoS defense system (DDS) : can block connection-based DoS attacks as well as attacks with legitimate content but bad behavior. It can also block both protocol attacks and rate-based attacks [55].

- Firewalls : along with a solid set of rules can bloc incoming traffic regarding ports, protocols, originating addresses, ... More complex attacks can still pass through a simple set of rules [40].

- Packet filtering : stations in the network can detect and dismiss packets with spoofed IP addresses reducing the effectiveness of reflected attacks [30].

Many additional defenses can be discussed and used to enhance this model. See the document [64] published by Peng et al. for a global survey of network defense mechanisms against DoS and DDoS problems.

At last, the properties of the leaf nodes has to be created. Their values are given by the user's evaluation. We tried to give coherent costs and probabilities to the attacks but we do not state that they are a good representation of real-life scenarios. The cost metric encompasses multiple aspects of the attack steps like the time period needed to achieve the step, the task's complexity, the technical skills required, ... The success probability metric takes in account the step's possibility to succeed (low, medium, high).

**Results**

The results of this use case are available in appendix B.3. Screenshots of the tool show the attack's translation into grammar and graphical representation.

Again, the resulted tree brings order into the DoS attack scenario with a readable graphical representation. This tree leans on the creation of shared children like the node "BuildBotnet". This attack step is used by multiple DoS attacks and the features of the tool will identify this node as a critical attacker action.

Regarding the attack paths computation, we can conclude that OR gates are problematic. The number of attack paths computed is 126077 for a total of 11 non redundant solutions (see section about reduction 5.6). This is a major performance concern as the tool computes a tremendous amount of non relevant solutions. The reduction section brings a solution to this problem and this solution is discussed in the areas of improvement section 7.1. However, this huge tree is still processed by the application exposing its potential.

This example demonstrates the interest of supplementary metrics as discusses in section 1.5. Alone, the cost metric is not expressive enough and the addition of damage, skill level or time concepts is a necessary upgrade. It is still possible to retrieve useful classification regarding the cost and the success probability.

## 6.4 Conclusion

Three main conclusions are highlighted by these use cases. (1) The grammar allows the modeling of quite complex scenarios and can still be enhanced. (2) The graphical tool displays readable and interactive trees in html format. Its computation is fluid and fast. (3) The attack paths computation time naturally explodes with large tree using multiple OR gates. This is a consideration to take into account while creating such large trees.

# Chapter 7

# Conclusion

This thesis aimed to propose a DAG-based attack tree model for system risk assessment along with a tool to create and analyse attack scenarios with this model. The project went through three different phases : (1) A research phase around the attack tree topic to understand the basis and the different existing models, (2) a conception phase to choose and adapt an attack tree model to the needs of our project and (3) a development phase for creating the tool and the desired functionalities.

*"What do we want to model ?"* The focus of this work is the description and representation of attack scenarios against systems to perform risk assessment. Risk assessment relies on a complete description of the attacker's actions and the defender's countermeasures in order to provide convincing results. The research around this topic phase brought the project toward a DAG-based model to represent the attack scenarios as we were looking for a highly visual model allowing extensions like shared children nodes and countermeasures. This type of structure adds a useful scalability property to the model because real systems are much more complex than the examples seen in the state of the art.

*"What model do we propose ?"* The proposed model integrates attack and defense modeling, supports order dependencies through the use of sequential AND gates, is currently specific to two types of numerical properties (cost and success probability) but can be easily enhanced with additional properties and finally proposes multiple added features with respect of the state of the art like the computation of attack traces under constraints based on the numerical properties of the nodes. This description refers to the classification table from the survey [48] published by Kordy et al.

*"How to develop the adapted tool ?"* We developed a lightweight python tool with the PyQt5 module. A clear interface invites the user to create and display attack scenarios with a clear grammar 4. An intuitive representation of attack

trees is achieved thanks to the Pyvis python module while other external python modules provide insightful quantitative information in terms of optimal attack paths and values. Furthermore, the tool proposes analysis features for the users to assess and compare different attack scenarios. As a last requirement, the tree structure scales well and the tool is resilient to user's errors.

The proposed tool is an additional use case to the attack tree modeling practice. The use cases described in section 6 exposes the benefits of the proposed graphical representation and the relevance of the analysis features. The last section of this document will expose ways to enhance and expand the tool.

## 7.1 Areas of improvement

As evoked during the document, the tree model and the implemented tool can be improved in many ways. This last section exposes some suggestions for further extensions.

**K/N gate :** As described in the SAT solving sections and the solution space reduction, the presence of OR gates creates redundant or non minimal attack paths. Indeed, if an attacker has two options to achieve its goal, he should prefer use only one solution based on metrics like the cost or the success probability. Along this consideration, redundant solutions negatively affect the tool in two ways : (1) The user is drowned in a massive amount of non relevant solutions and (2) the SAT solver computes the same amount of non suitable solutions.

The implementation of "K out of N" gate would tackle this problem. With this gate, it becomes possible to express the fact that the attacker would choose only one action between multiple ones instead of performing them the same way an OR gate would do. Note that a "1 out of 2" gate is available thanks to the existing XOR gate. This gate is satisfied if an odd number of input variables are false. Hence, this gate only mimics a "K out of N" gate in two inputs situations.

**Countermeasure metrics :** It is currently not possible to assign metrics to the countermeasures. We were not convinced about the way to process them and we chose to let the topic aside. Indeed, countermeasures can be handled in various ways. The first way to proceed them is to simply say that an action node with a countermeasure is inaccessible to the attacker and no attack trace can be successful with this action. The tool is able to model this kind of behaviour with the variable fixation 5.7 feature. However, this is not a really nice way to illustrate real-life scenarios. Another way to proceed those countermeasure metrics is to make their numerical values to influence the numerical metrics of the system. This is not an easy task as each metrics has a proper meaningful way to be processed. This topics deserves to be explored as it would enhance the expressiveness of the actual model.

**Tseitin :** A better usage of Tseitin transformation could greatly improve the way our implementation was designed. By keeping the auxiliary variables corresponding to the intermediary nodes we could make use of them to show the results in a better way (coloring the attack traces and checking if they are enabled or not in the solution would become trivial). Another interesting feature would be to use the auxiliary variables to analyze the sub-trees that they represent.

**Attack tree synthesis :** A recent document published by Pinchinat et al. [65] proposes an algorithm to do attack tree synthesis. Their goal is to build a tree (if any) based on an observed set of attack traces. An integration of such tool to the actual implementation would allow users to create trees from the specifications of their system and also from a set of attack traces.

**Optimization and global implementation :** As our implementation was a starting point in the discovery and analysis of the main principles behind Attack Trees, some parts may need to be redesigned and optimized to seek for better performance. After this thesis, the knowledge acquired could help us to create a more specialized tool focused on quicker and more effective computation. As an example we can cite SAT and SMT solvers that have many existing configurations that may be more adequate to our problem.

# Bibliography


[1] Ctypes python documentation 3.10.4. https://docs.python.org/3/library/ctypes.html.

[2] Directed ayclic graph overview & use cases. https://hazelcast.com/glossary/directed-acyclic-graph/.

[3] Distributed denial of service attack (ddos) definition. https://www.imperva.com/learn/ddos/ddos-attacks/.

[4] Eir's d1000 modem is wide open to being hacked. https://devicereversing.wordpress.com/2016/11/07/eirs-d1000-modem-is-wide-open-to-being-hacked/i.

[5] Json-c - a json implementation in c. https://github.com/json-c/json-c/wiki.

[6] Pyvis - a python library for visualizing networks. https://pyvis.readthedocs.io/en/latest/.

[7] The reaper iot botnet has already infected a million networks, december 2018. https://www.wired.com/story/reaper-iot-botnet-infected-million-networks/.

[8] The z3 theorem prover. https://github.com/Z3Prover/z3.

[9] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224, 2002.

[10] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*, pages 1093–1110, 2017.

[11] Zaruhi Aslanyan and Flemming Nielson. Pareto efficient solutions of attack-defence trees. In *International Conference on Principles of Security and Trust*, pages 95–114. Springer, 2015.

[12] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. 2009.

[13] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.

[14] Maxime Audinot, Sophie Pinchinat, and Barbara Kordy. Is my attack tree correct? In *European Symposium on Research in Computer Security*, pages 83–102. Springer, 2017.

[15] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–122, 2019.

[16] Udi Ben-Porat, Anat Bremler-Barr, and Hanoch Levy. Vulnerability of network mechanisms to sophisticated ddos attacks. *IEEE Trans. Computers*, 62(5):1031–1043, 2013.

[17] Leyla Bilge and Tudor Dumitraş. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844, 2012.

[18] Magnus Björk. Successful sat encoding techniques. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):189–201, 2011.

[19] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. $\nu$z-an optimizing smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.

[20] Eric J Byres, Matthew Franz, and Darrin Miller. The use of attack trees in assessing vulnerabilities in scada systems. In *Proceedings of the international infrastructure survivability workshop*, pages 3–10. Citeseer, 2004.

[21] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[22] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34. IEEE, 2007.

[23] Evan Damon, Julian Dale, Evaristo Laron, Jens Mache, Nathan Land, and Richard Weiss. Hands-on denial of service lab exercises using slowloris and rudy. In *proceedings of the 2012 information security curriculum development conference*, pages 21–29, 2012.

[24] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[25] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[26] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[27] Heidi E Dixon, Matthew L Ginsberg, and Andrew J Parkes. Generalizing boolean satisfiability i: Background and survey of existing work. *Journal of Artificial Intelligence Research*, 21:193–243, 2004.

[28] Wesley Eddy et al. Tcp syn flooding attacks and common mitigations. Technical report, RFC 4987, August, 2007.

[29] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[30] Paul Ferguson and Daniel Senie. rfc2827: network ingress filtering: defeating denial of service attacks which employ ip source address spoofing, 2000.

[31] Marlon Fraile, Margaret Ford, Olga Gadyatskaya, Rajesh Kumar, Mariëlle Stoelinga, and Rolando Trujillo-Rasua. Using attack-defense trees to analyze threats and countermeasures in an atm: a case study. In *IFIP Working Conference on The Practice of Enterprise Modeling*, pages 326–334. Springer, 2016.

[32] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT Workshop 2015*, 2015.

[33] Weiwei Gong and Xu Zhou. A survey of sat solver. In *AIP Conference Proceedings*, volume 1836, page 020059. AIP Publishing LLC, 2017.

[34] Marco Gribaudo, Mauro Iacono, and Stefano Marrone. Exploiting bayesian networks for the analysis of combined attack trees. *Electronic notes in theoretical computer science*, 310:91–111, 2015.

[35] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[36] Jin B Hong, Dong Seong Kim, Chun-Jen Chung, and Dijiang Huang. A survey on the usability and practical applications of graphical security models. *Computer Science Review*, 26:1–16, 2017.

[37] Ross Horne, Sjouke Mauw, and Alwen Tiu. Semantics for specialising attack trees based on linear logic. *Fundamenta Informaticae*, 153(1-2):57–86, 2017.

[38] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[39] Terrance R Ingoldsby. Attack tree-based threat risk analysis. *Amenaza Technologies Limited*, pages 3–9, 2010.

[40] Sotiris Ioannidis, Angelos D Keromytis, Steve M Bellovin, and Jonathan M Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 190–199, 2000.

[41] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international sat solver competitions. *Ai Magazine*, 33(1):89–92, 2012.

[42] Ravi Jhawar, Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Rolando Trujillo-Rasua. Attack trees with sequential conjunction. In *IFIP International Information Security and Privacy Conference*, pages 339–353. Springer, 2015.

[43] Aivo Jürgenson and Jan Willemson. Serial model for attack tree computations. In *International Conference on Information Security and Cryptology*, pages 118–128. Springer, 2009.

[44] Maurice Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

[45] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[46] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Foundations of attack–defense trees. In *International Workshop on Formal Aspects in Security and Trust*, pages 80–95. Springer, 2010.

[47] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Attack–defense trees. *Journal of Logic and Computation*, 24(1):55–87, 2014.

[48] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. Dag-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer science review*, 13:1–38, 2014.

[49] Barbara Kordy and Wojciech Wideł. On quantitative analysis of attack–defense trees with repeated labels. In *International Conference on Principles of Security and Trust*, pages 325–346. Springer, 2018.

[50] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining anomalies in feature models. *ACM SIGPLAN Notices*, 52(3):132–143, 2016.

[51] Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. Quantitative attack tree analysis via priced timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 156–171. Springer, 2015.

[52] Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. A review of attack graph and attack tree visual syntax in cyber security. *Computer Science Review*, 35:100219, 2020.

[53] Henry CJ Lee and Vrizlynn LL Thing. Port hopping for resilient networks. In *IEEE 60th Vehicular Technology Conference, 2004. VTC2004-Fall. 2004*, volume 5, pages 3291–3295. IEEE, 2004.

[54] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1033–1050, 2016.

[55] Xin Liu, Xiaowei Yang, and Yanbin Lu. To filter or to authorize: Network-layer dos defense against multimillion-node botnets. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 195–206, 2008.

[56] Joao Marques-Silva. Practical applications of boolean satisfiability. 2008.

[57] Mark Mateski, Cassandra M Trevino, Cynthia K Veitch, John Michalski, J Mark Harris, Scott Maruoka, and Jason Frye. Cyber threat metrics. *Sandia National Laboratories*, page 30, 2012.

[58] Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In *International Conference on Information Security and Cryptology*, pages 186–198. Springer, 2005.

[59] Edward J McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[60] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[61] 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*, pages 337–340. Springer, 2008.

[62] National Electric Sector Cybersecurity Organization Resource (NESCOR). Analysis of selected electric sector high risk failure scenarios. `https://smartgrid.epri.com/doc/NESCOR%20Detailed%20Failure%20Scenarios%20v2.pdf`, 2015.

[63] Ramamohan Paturi, Pavel Pudlák, Michael E Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. *Journal of the ACM (JACM)*, 52(3):337–364, 2005.

[64] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Computing Surveys (CSUR)*, 39(1):3–es, 2007.

[65] Sophie Pinchinat, François Schwarzentruber, and Sébastien Lê Cong. Library-based attack tree synthesis. In *International Workshop on Graphical Models for Security*, pages 24–44. Springer, 2020.

[66] Mukul R Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[67] Steven David Prestwich. Cnf encodings. *Handbook of satisfiability*, 185:75–97, 2009.

[68] PyQT. Pyqt reference guide.

[69] John H Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[70] Christian Rossow. Amplification hell: Revisiting network protocols for ddos abuse. In *NDSS*, pages 1–15, 2014.

[71] Arpan Roy, Dong Seong Kim, and Kishor S Trivedi. Attack countermeasure trees (act): towards unifying the constructs of attack and defense trees. *Security and communication networks*, 5(8):929–943, 2012.

[72] B. Schneier. Attack trees, dr. dobb's journal of software tools, 24, 21–29, 1999. 1999.

[73] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148. IEEE, 2006.

[74] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer networks*, 51(2):456–479, 2007.

[75] Robert Shirey. Internet security glossary, 2000.

[76] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

[77] Maurice H ter Beek, Axel Legay, Alberto Lluch Lafuente, and Andrea Vandin. Quantitative security risk modeling and analysis with risqflan. *Computers & Security*, 109:102381, 2021.

[78] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[79] Ruchi Vishwakarma and Ankit Kumar Jain. A survey of ddos attacking techniques and defence mechanisms in the iot network. *Telecommunication systems*, 73(1):3–25, 2020.

[80] Wojciech Wideł, Maxime Audinot, Barbara Fila, and Sophie Pinchinat. Beyond 2014: Formal methods for attack tree–based security modeling. *ACM Computing Surveys (CSUR)*, 52(4):1–36, 2019.

[81] Fekadu Yihunie, Eman Abdelfattah, and Ammar Odeh. Analysis of ping of death dos and ddos attacks. In *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–4. IEEE, 2018.

[82] Liu Yu, Kailiang Chen, Yue Chang, A Chen, Qidi Yin, and Hangwei Zhang. A new correlation model of iot attack based on attack tree. In *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/-CyberSciTech)*, pages 930–935. IEEE, 2021.

[83] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *International conference on computer aided verification*, pages 17–36. Springer, 2002.

# Appendix A

# Screens and features

This appendix will describe the different screens and features available in the application. See chapter 5 for the description of the presented features. The example used in this appendix refers to the example used in section 4.2.

## A.1 Text window

The text window of the tool is divided into four components :

- **Tree grammar space :** This large editable area is the place to input the tree in the grammar format.
- **Import button :** This button allows the user to import the grammar of a tree from a *txt* file.
- **Create button :** This button will create the *json* file describing the input tree.
- **File name :** This field is the name given to the *json* file created.

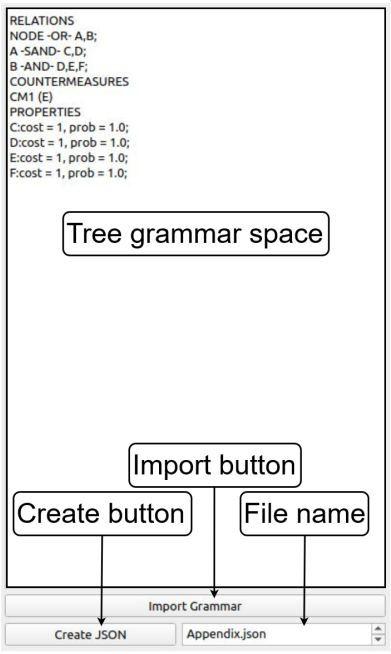



Figure A.1: Text window

## A.2 Display window

The display window of the tool is divided into four components :

- **Tree display window :** This window plots the interactive graphical representation of the attack tree.
- **Import button :** This button allows the user to import the *json* file of a tree to display it on the screen.
- **Reload button :** This button will recompute the displayed tree. This feature is useful to apply some option from the feature column. A.3
- **File name :** This text field contains the path of the *json* file into the file system.

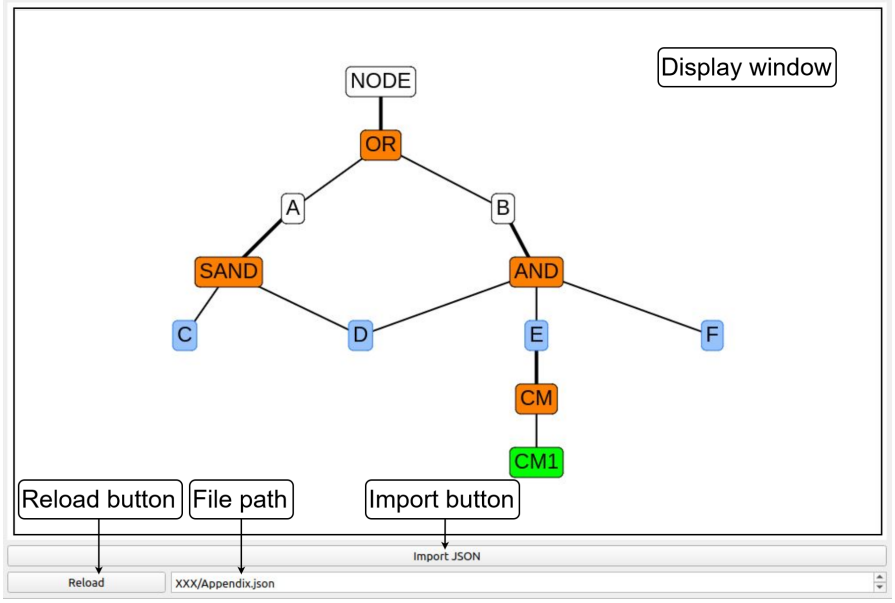



Figure A.2: Tool display window

## A.3 Feature column

See chapter 5 for a deeper comment about these features.

### A.3.1 Solver

This first component of the feature column is used to display the solutions. The solutions are computed by the SAT solver which is the second component of the feature column A.3.2. This is the list of the different options provided by this component :

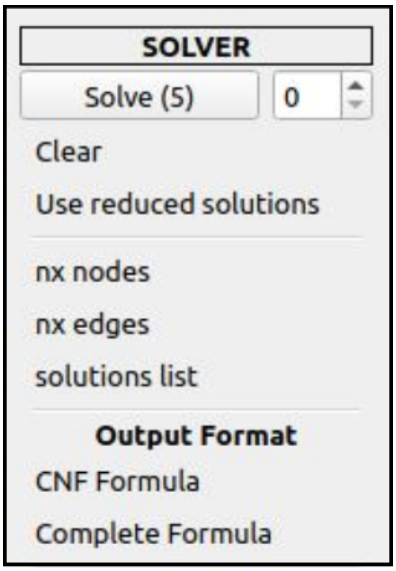



Figure A.3: Tool column feature

- **Solve(X) :** This button displays the solutions of the attack on the tree. The number of available solutions to display is the number between parenthesis (here 5). The number of available solutions can be capped by the SAT solver component A.3.2. The selector on the right permits to select the chosen attack path. Once pressed, the attack path is displayed on the tree. The red nodes are the actions selected by the attack path to achieve the root node.
- **Clear :** This button resets the displayed tree and removes the solutions displayed on it.
- **Use reduced solutions :** This button removes the redundant solutions from the solution list. See section 5.6 for a deep comment about this feature.
- **nx nodes / nx edge :** These buttons respectively display the Networkx nodes and Networkx edges of the tree. This mainly has a debugging use.
- **solutions list :** This button displays the list of solutions to the tree. This list can be reduced with the reduction option.
- **Output Format :** These two buttons modify the display format of the boolean formula in the solution window A.4. The first button sets the display format to the CNF format 2.3.2 while the second button sets the display format to the complete boolean formula. These are the two equivalent formulas of the appendix example :

$$Compact \ : \ ( \ ( \ C \ \& \ D \ ) \ | \ ( \ D \ \& \ E \ \& \ F \ ) \ )$$
$$\Leftrightarrow$$
$$CNF \ : \ D \ \& \ ( \ C \ | \ D \ ) \ \& \ ( \ C \ | \ E \ ) \ \& \ ( \ C \ | \ F \ ) \ \& \ ( \ D \ | \ E \ ) \ \& \ ( \ D \ | \ F \ )$$

### A.3.2 SAT solver

The second component of the feature column is the SAT solver. A few options are available and listed here :

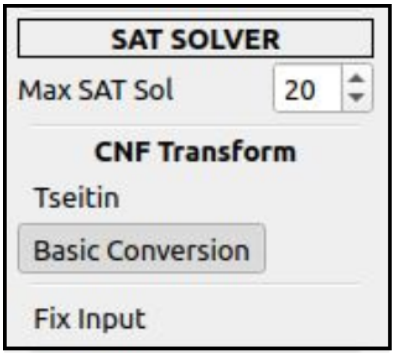



Figure A.4: Tool column feature

- **Max SAT Sol :** This selector allows the user to select the maximum number of solutions the SAT solver must calculate. This is useful for the computation of large tree as the number of solutions increase quickly with the number of variable and OR gates. Writing "-1" in the numerical field lets the SAT solver compute all the solutions of the tree. Use the "Reload" button A.2 to apply the changes. By default, the number of solutions is capped at 20.
- **Tseitin / Basic Conversion :** These options allow the user to choose between the Tseitin CNF translation and a classic CNF translation. See section 2.3.2 for a deep comment on these options.
- **Fix input :** This tool allow the user to fix the different variables of the system. It is possible to force the variables to be either *true* or *false* and recompute the solutions under these conditions. It is possible to fix the countermeasures of the tree which automatically set to false all the concerned nodes. This is a useful feature to understand the impact of these countermeasures on the tree.

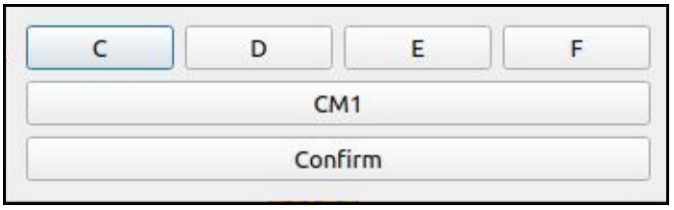



Figure A.5: Fix input window

### A.3.3 SMT solver

The third component of the feature column is the SMT solver. Using attack tree with defined costs and probabilities, the SMT solver can compute all the attacks with total cost lower than the input number and the all the attack with a success probability greater than the input probability. To do so, simply press the button once the numerical constraint is set up. The solutions returned will have lower cost (or higher success probability) than the input value. If no numerical value is inserted, the best attack path and its value are returned as the only best solution. The theoretical concept behind this feature is detailed in section 2.4.

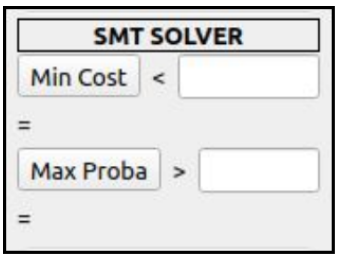



Figure A.6: Tool column feature

### A.3.4 Other modules

#### Comparison

The comparison module asks the user to input two attack trees in the *json* file format and opens a new window. See the example provided in section 5.8 to have a view of this feature.

#### Nodes Frequencies

The node frequencies module displays the tree in term of the use frequency of every node in every possible attack. Note that the graph is based only on the reduced solutions if this option is activated in the Solver component A.3.1. Consider limiting the solution space to avoid disruption form the redundancies. See the section 5.9 for comments on this feature. Here is the frequency diagram of the example. It highlights the fact that node D is mandatory for any attack path of the tree.

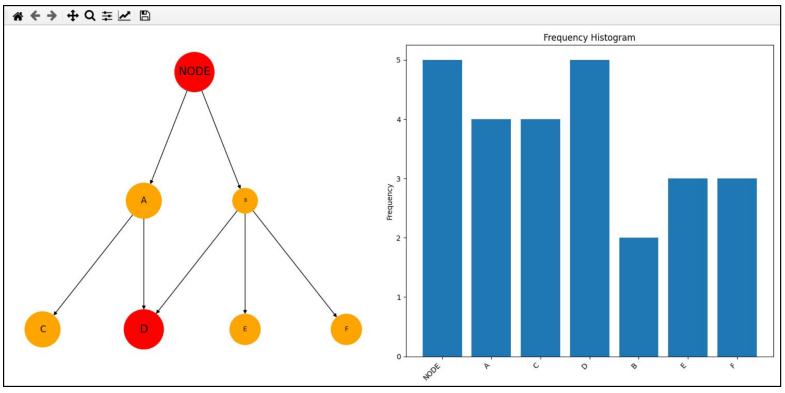



Figure A.7: Frequency Diagram tool

**Random Tree**

This module is the input for the random tree generator described in section 4.7.The three digital inputs are as follows :

- **Argument 1** : maxdepth. A relatively small number.
- **Argument 2** : branching_factor. A relatively small number.
- **Argument 3** : complexity. A number between 1 and 5 inclusive.

Upon clicking on the "Random Tree" button, the grammar is written in the text window and the *json* file is directly generated with the file name given in the file name field.


Figure A.8: Random tree input

## A.4 Solution window

The solution window outputs the truth table of the selected solution. For each boolean variable, the table specifies its value. Additional variables can be displayed with the Tseitin CNF transformation as it creates intermediate variables. Here is the truth table of an attack path of the previous example A.2 :

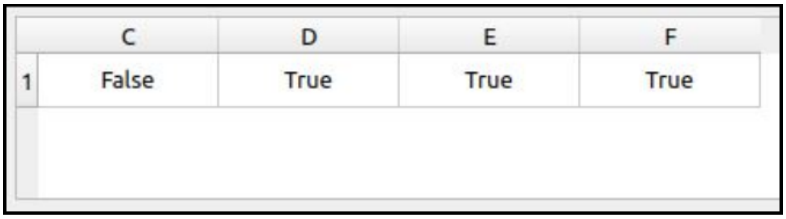



Figure A.9: Example of solution window

# Appendix B

# Use cases screenshots

### Mirai use case

```
RELATIONS
SpreadMirai -SAND- FindDevice,Penetrate,AdvertiseMaster,Hide;
FindDevice -SAND- NetworkScan,GetIoTResponse;
NetworkScan -OR- Telnet,HTTPS,FTP,SSH,CWMP;
Penetrate -OR- GuessCredentials,FactoryCredential;
AdvertiseMaster -SAND- Connect,LoadMalware;
Hide -AND- BlocAdminPorts,EraseCompetitor;

COUNTERMEASURES
Strong password (GuessCredentials, FactoryCredential)
Limited Remote Access (GetIoTResponse)
Isolation boundaries (LoadMalware,BlocAdminPorts)
Scan Notification (NetworkScan)

PROPERTIES
Telnet:cost = 0,prob = 1,0;
HTTPS:cost = 0,prob = 1,0;
FTP:cost = 0,prob = 1,0;
SSH:cost = 0,prob = 1,0;
CWMP:cost = 0,prob = 1,0;
GetIoTResponse:cost = 0,prob = 0,001;
FactoryCredential:cost = 0,prob = 0,001;
GuessCredentials:cost = 0,prob =0,01;
Connect:cost = 0,prob = 1.0;
LoadMalware:cost = 1,prob = 1.0;
BlocAdminPorts:cost = 0,prob = 1.0;
EraseCompetitor:cost = 5,prob = 0.5;
```

Figure B.1: Mirai malware grammar attack tree

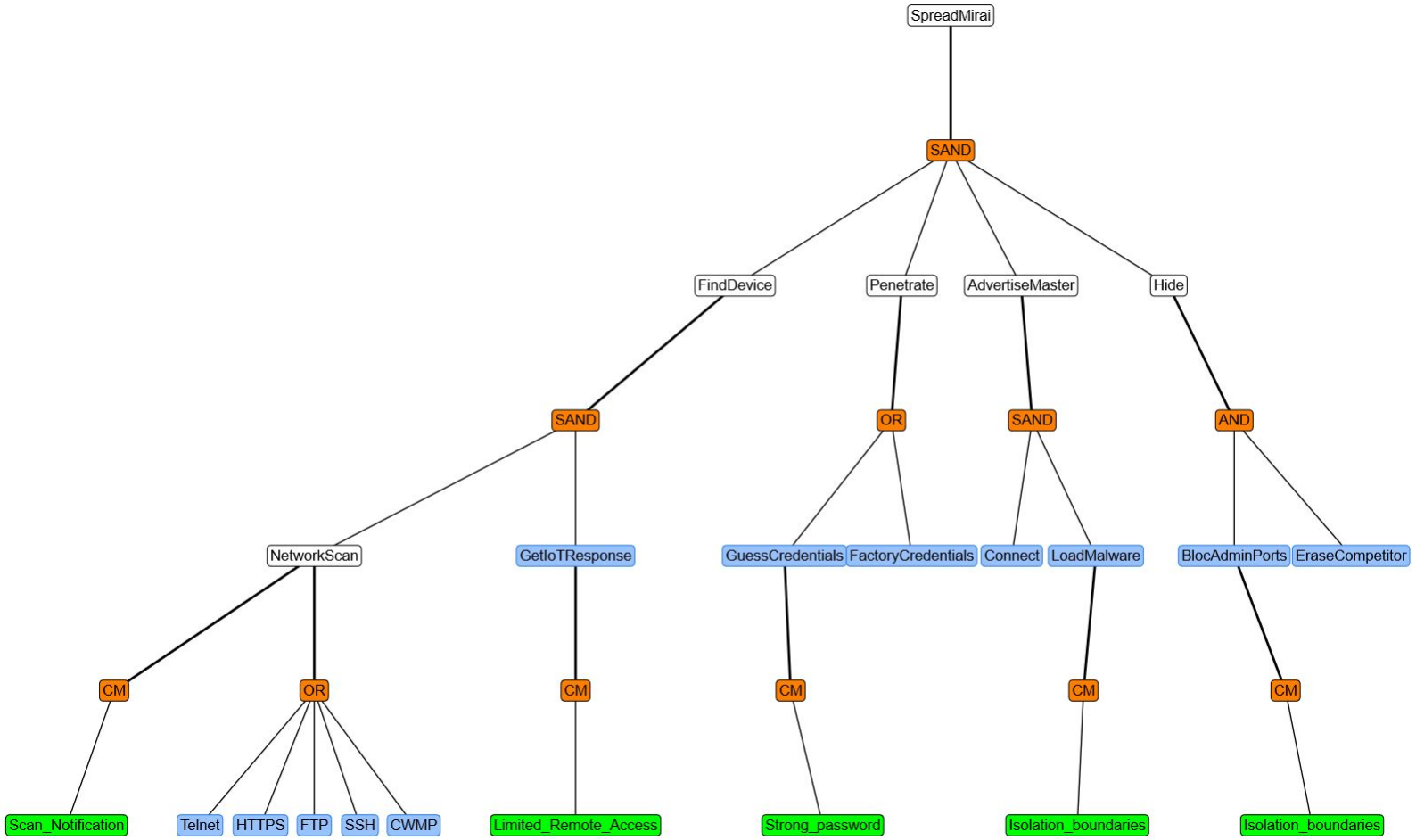



Figure B.2: Mirai malware graphical tree

### DoS use case

```
RELATIONS
DoSAttack -OR- VolumeAttack,ProtocolAttack,ApplicationAttack,ZeroDayAttack;
VolumeAttack -OR- UDPflood,ICMPflood,DRDoS,AmplifiedDRDoS;
ProtocolAttack -OR- SYNflood,PingofDeath,Slowloris,PerpetualEcho;
ApplicationAttack -OR- HTTPflood,SophisticatedDDoS;
UDPflood -SAND- BuildBotnet,UDPattack;
ICMPflood -SAND- BuildBotnet,ICMPattack;
DRDoS -SAND- BuildBotnet,SpoofVictimAdress,ReflectedAttack;
AmplifiedDRDoS -SAND- BuildBotnet,SpoofVictimAdress,AmplifyAttack;
SYNflood -SAND- SendManySYN,KeepConnectionsAlive;
PingofDeath -SAND- ForgePacket,SendtoVictim;
Slowloris -SAND- OpenPOSTtransmissions,InfiniteSlowDataSend;
PerpetualEcho -SAND- ForgeEchoRequest,SendtoVictim;
HTTPflood -SAND- ForgeHTTPrequest,ExhaustVictimResources;

COUNTERMEASURES
Upstream filtering (ProtocolAttack)
Blackholing (VolumeAttack)
DDS (VolumeAttack,ProtocolAttack)
Firewalls (ProtocolAttack)

PROPERTIES
BuildBotnet:cost = 10,prob = 1,0;
UDPattack:cost = 0,prob = 1,0;
ICMPattack:cost = 0,prob = 1,0;
SpoofVictimAdress:cost = 0,prob = 1,0;
ReflectedAttack:cost = 0,prob = 0,8;
AmplifiedDRDoS:cost = 0,prob = 0,8;
SendManySYN:cost = 0,prob = 1,0;
KeepConnectionsAlive:cost = 1,prob = 1,0;
ForgePacket:cost = 1,prob = 0,8;
SendtoVictim:cost = 0,prob = 0,8;
OpenPOSTtransmissions:cost = 0,prob = 1,0;
InfiniteSlowDataSend:cost = 1,prob = 1,0;
ForgeEchoRequest:cost = 1,prob = 0,8;
ForgeHTTPrequest:cost = 1,prob = 0,8;
ExhaustVictimResources:cost = 0,prob = 0,8;
SophisticatedDDoS:cost = 5,prob = 0,2;
ZeroDayAttack:cost = 20,prob = 1,0;
```

Figure B.3: DoS attack grammar

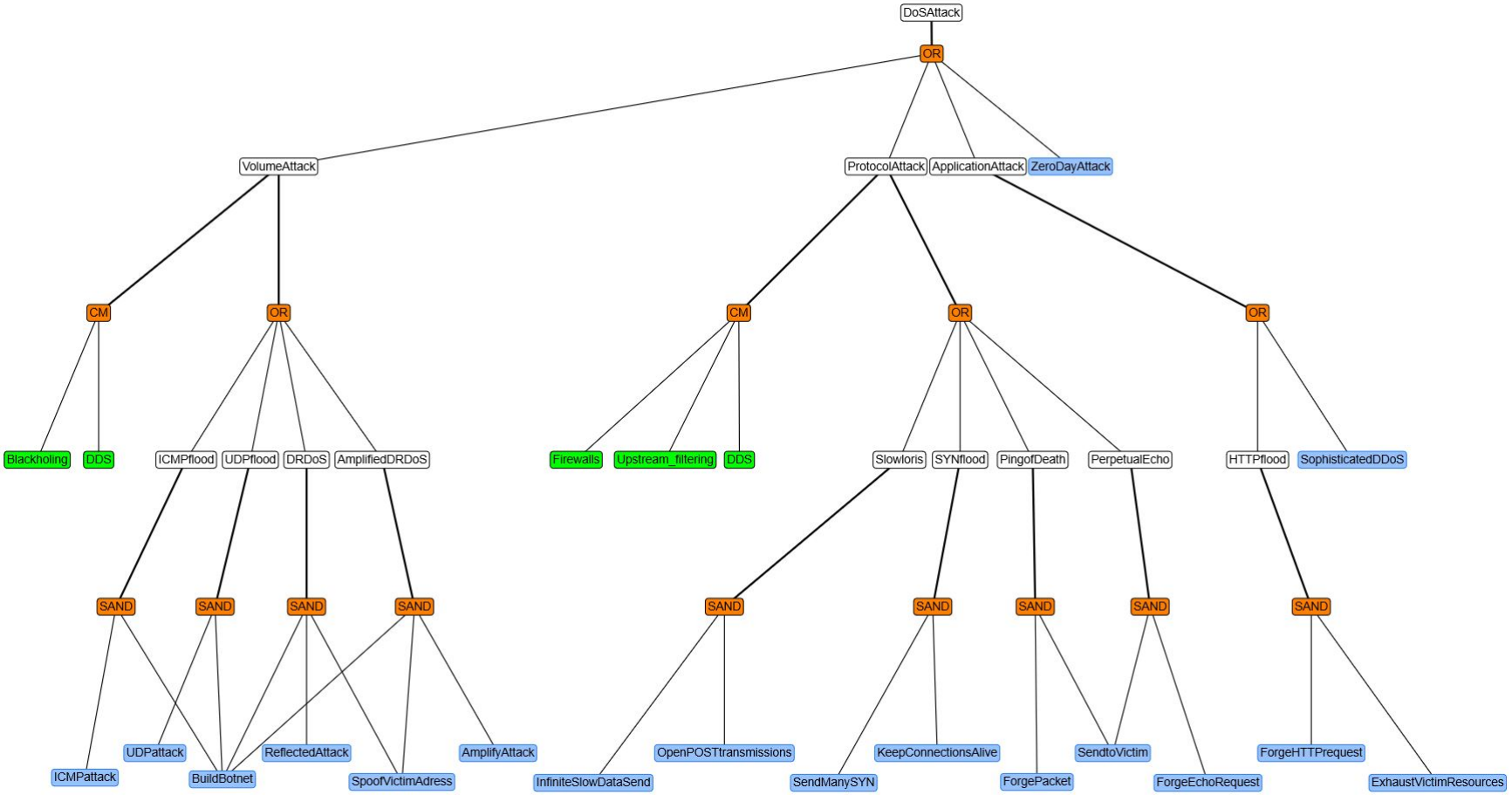



Figure B.4: DoS attack graphical tree

# Appendix C

# Data files and algorithms

The *json* save file of the tree example present in section 4.2.

```
{
    "Action":"NODE",
    "Type":"OR",
    "Child":[
        {
            "Action":"A",
            "Type":"SAND",
            "Child":[
                {
                    "Action":"C",
                    "Type":"LEAF",
                    "Cost":1,
                    "Prob":1.0
                },
                {
                    "Action":"D",
                    "Type":"LEAF",
                    "Cost":1,
                    "Prob":1.0
                }
            ]
        },
        {
            "Action":"B",
            "Type":"AND",
            "Child":[
                {
                    "Action":"D",
                    "Type":"LEAF",
                    "Cost":1,
                    "Prob":1.0
                },
                {
                    "Action":"E",
                    "Type":"LEAF",
                    "Cost":1,
                    "Prob":1.0,
                    "CM":[
                        {
                            "CMtitle":"CM1",
```

```
                            "CMcost":0,
                            "CMprob":1.0
                        }
                    ]
                },
                {
                    "Action":"F",
                    "Type":"LEAF",
                    "Cost":1,
                    "Prob":1.0
                }
            ]
        }
    ]
}
```

**Algorithm 5:** Tseitin CNF transformation

```
Data: Formula expression
Result: CNF expression, set of variables,
def tseitin(formula):
    // New substitution variables notation :  %i
    recur_formula(formula, [], , 0, set()) // Create a list of auxiliary variables
       from sub-expressions and replace them in the formulas
    cnf_clauses = [] // Final conjunction list
    cnf_clauses ← add the whole formula substitution
    for var, expr : the new variables and their sub-expressions do
        if expr is 'And' then
            cnf_clauses ← add each [argument, Not(var)] clauses
            cnf_clauses ← add [var and every Not(argument)]
        else if expr is 'Or' then
            cnf_clauses ← add each [Not(argument), var] clauses
            cnf_clauses ← add [Not(var) and every arguments]
        else if expr is 'Xor' then
            for c in every possible boolean combinations do
                list_1 = variables combination using c : True is X, False is ¬X
                if even nbr of True values then
                    if even nbr of variables then  list_1 ← add Not(var)
                    else  list_1 ← add var
                else
                    if even nbr of variables then  list_1 ← add var
                    else  list_1 ← add Not(var)
                cnf_clauses ← add list_1
        else if expr is 'Not' then
            if expr is 'Nand' then
                cnf_clauses ← add each [argument, var] clauses
                cnf_clauses ← add [Not(var) and every Not(argument)]
            else if expr is 'Nor' then
                cnf_clauses ← add each [Not(argument), Not(var)] clauses
                cnf_clauses ← add [var and every arguments]
            else if expr is 'Xnor' then
                for c in every possible boolean combinations do
                    list_1 = variables combination using c : True is X, False is ¬X
                    if even nbr of True values then
                        if even nbr of variables then  list_1 ← add var
                        else  list_1 ← add Not(var)
                    else
                        if even nbr of variables then  list_1 ← add Not(var)
                        else  list_1 ← add var
                    cnf_clauses ← add list_1
            else  cnf_clauses ← add [Not(argument), Not(var)] and [argument, var]
    cnf_clauses = [Or(*l) for l in cnf_clauses] // Clauses disjunction conversion
    cnf_clauses = And(*cnf_clauses) // Formula conjunction conversion
    return cnf_clauses, set of variables, corresponding auxiliary variables
```

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**