Bachelor thesis
Computing Science

Radboud University

Analysis of Attack Trees: fast algorithms for subclasses

Author:
Lars Kuipers
s1005864

First supervisor/assessor:
Prof. dr. Marielle Stoelinga
m.stoelinga@cs.ru.nl

Second assessor:
Prof. dr. Frits Vaandrager
f.vaandrager@cs.ru.nl

June 28, 2020

Abstract

Attack trees are a graphical security model which allow to systematically categorize the different ways in which a system can be attacked. They are appealing to practitioners, yet also useful for tool builders attempting to partially automate the threat analysis. Therefore, attack trees are really useful for the security community for threat analysis. It helps to improve the security of a system within a company by analysing the ways in which the system can be attacked. Thus, it is important to have efficient analysis methods to be able to make great use of these attack trees. The problem, however, is that there are several extensions of attack trees which result in different types of attack trees. Some analysis methods work for one type, but do not work for other types. We go further into two of these types and describe fast algorithms to analyse those types of attack trees. For the first type, classical attack trees with possibly SAND gates, we show how the already known bottom up method works and we extend it for the addition of the SAND gate. For the second type of attack tree, attack trees with shared subtrees but no SAND gates, we explain how we can exploit the structure of the attack tree and analyse them by using binary decision diagrams. An attack tree will be converted to a binary decision diagram, which will then be used to analyse different properties of the attack tree. This results in several algorithms for different attributes to be able to analyse different properties of the attack tree.

Contents

1 Introduction **3**
1.1 The problem . . . 3
1.2 Attack trees . . . 4
1.3 Our contribution . . . 7
1.4 Thesis overview . . . 8

2 Preliminaries **9**
2.1 Attack Trees . . . 9
2.1.1 Attack tree structure . . . 10
2.1.2 Attack tree definition . . . 11
2.1.3 Attack tree semantics . . . 13
2.1.4 Metrics . . . 16
2.2 Binary decision diagrams . . . 19
2.2.1 BDD definition . . . 19
2.2.2 BDD reduction . . . 22
2.2.3 The impact of variable ordering . . . 23

3 Related Work on Attack Tree Analysis **24**
3.1 Semantics . . . 24
3.2 Analysis methods . . . 25
3.3 Application . . . 26

4 Analysis of attack trees **27**
4.1 Overview . . . 27
4.1.1 Types of attack trees . . . 27
4.1.2 Results . . . 28
4.2 TREES with SAND gates . . . 29
4.3 Attack trees with shared subtrees . . . 38
4.3.1 Why BDD method? . . . 38
4.3.2 Translating AT to BDD . . . 39
4.3.3 Methods for different attributes . . . 41

5 Conclusions **65**

6 Acknowledgements **67**

Chapter 1

Introduction

1.1 The problem

The world is getting more digital by the day. This brings a lot of advantages along, but also some major risks. The vulnerability of digital systems to cyber attacks is one of these major threats [2], varying from DoS and DDoS attacks to phishing or password attacks. There is a need to understand all the different ways in which a system can be attacked, so that appropriate countermeasures can likely be designed to thwart those attacks. Often security decisions are made informally, e.g. by brainstorming. More structures approaches are based on spreadsheets and technical standards.

Graphical security models provide a useful method to represent and analyse security scenarios that examine vulnerabilities of systems and organizations. The great advantage of graph-based approaches lies in combining user friendly, intuitive, visual features with formal semantics and algorithms that allow for qualitative and quantitative analyses [11]. Over the last decades a lot of such graphical models have been developed. A historical overview on existing graph-based approaches is given in [14]. A good overview of attack tree like models has been given in [11]. It describes the *forest* of the trees, among which we find the attack tree which will be discussed in this thesis.

This thesis focuses on attack trees, introduced by Schneier in [15]. Attack trees form a convenient way to systematically categorize the different ways in which a system can be attacked. The graphical, structured tree notation is appealing to practitioners, yet also very promising for tool builders attempting to partially automate the threat analysis. Therefore attack trees are of great interest to the security community for threat analysis [13]. For example a company which works with a lot of sensitive data. They must be sure their system is as safe as possible to prevent any leaks. Attacks trees can help them fulfilling this task by giving a better overview of how a system might be attacked and what the weak points in the system are. Furthermore, attack trees are also part of the security standards.

1.2 Attack trees

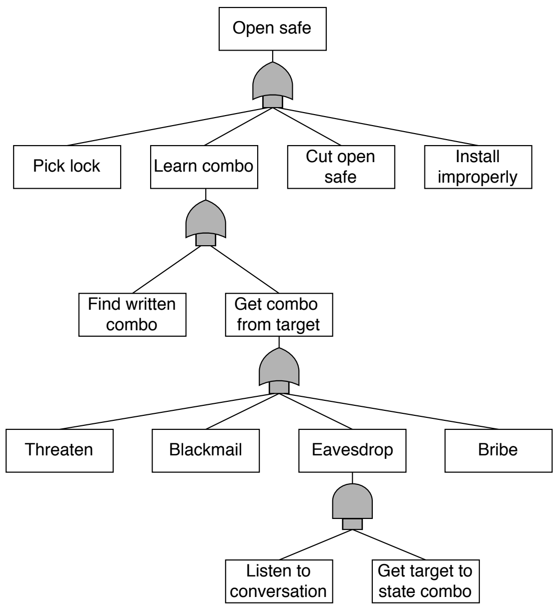


Figure 1.1: Classic attack tree

In short, an attack tree is a tree in which the nodes represent (sub)attacks, see Figure 1.1. The root node of the tree is the global goal of an attacker. Children of a node are refinements of this goal, and leaves therefore represent attacks that can no longer be refined. Such a refinement can be conjunctive, via AND gates, or disjunctive by using OR gates. Figure 1.1 shows an example of an attack tree. The root node is the goal of the attack. This goal can be refined further into sub attacks using gates. The basic attack steps of this attack tree are the leaves, for example 'Pick lock'. Those are the actual attacks and cannot be refined any further. These basic attack steps are also the nodes that contain all attribute values which an attack tree has. As introduced by Schneier [15], attack trees only had OR- and AND-gates and were real trees. However, there are two extensions of these attack trees which we will also cover in this thesis. The first one is shared subtrees, which means that attack trees are directed acyclic graphs rather than real trees, an example of this can be found in Figure 1.2. The second extension is the addition of another gate, which is the sequential version of the AND gate: the SAND gate. This is useful, because temporal order is often an important aspect in attacks. An example of this can be found in Figure 1.3. Attack trees will be explained more detailed in Section 2.1.

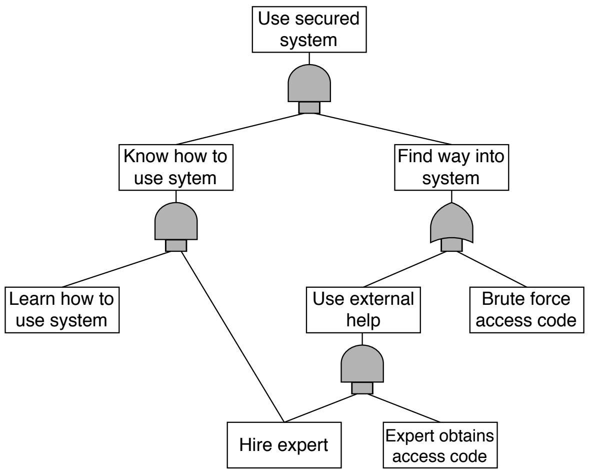


Figure 1.2: Attack tree with a shared subtree. The node 'Hire expert' can be reached in multiple ways

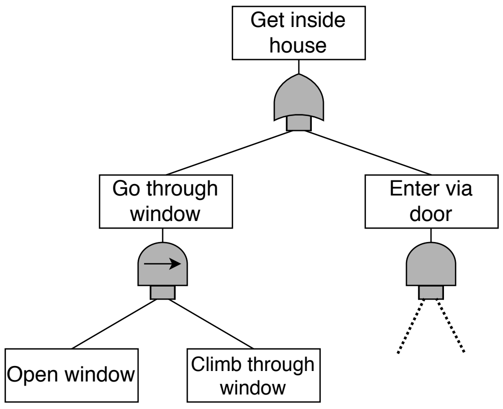


Figure 1.3: Part of an attack tree including a SAND gate. To go through the window, first the window needs to be opened and then one can climb through the window

In summary, attack trees are very useful to model cyber attacks on a system, but to make good use of attack trees they need to be analysed. Attack trees themselves only show the ways how a system can be attacked, but by analysing them we get to know more properties of a system, such as the weaknesses. The main reason for AT analysis therefore also is: find (cost-) effective counter measures against the attack on a system. However, there are many different types of attack trees, i.e. different gates, different structures (tree vs. DAG). Some of which we are able to analyse efficiently, but others are lacking efficient analysis methods. For example, the classic attack trees have fairly good methods as they can be analysed using the bottom up method. However, attack trees with shared subtrees and especially if they also include SAND gates are way harder to analyse, because they add another layer of complexity. Besides the fact that all children have to be executed, the order of the execution of the children is also a factor now. This makes for extra checks and furthermore changes the structure properties of the attack tree, because there is a new gate in it. There has been done a lot of research on analysing attack trees, some of which we describe in Chapter 3. Being able to understand attack trees better significantly helps to improve the security, for example to come up with effective countermeasures. Therefore it is very important that efficient analysis methods will be available. This brings us to our research question.

How can we make attack tree analysis more efficient by exploiting structural properties of the attack trees?

Attack tree analysis is important for several reasons. It helps us to find useful properties of different possible attacks to come up with countermeasures. Examples are the cheapest or the fastest attack, or even the probability that a system is attacked. With this information, we can understand what the attack goals are, which attacks are more likely to occur, determine the weak places in a system, understand where to best spend a security budget etc. [15]. We consider two types of analysis in this thesis.

1. A bottom up method where values at the leaves of the attack will be propagated towards the root node of the tree to find a certain property of the attack tree, for example the cheapest attack in the attack tree.

2. The second analysis method will be with the use of binary decision diagrams. It is the so called BDD method, where an attack tree is first converted to a binary decision diagram which will then be used to analyse properties of the attack tree.

We sometimes use the bottom up method and sometimes use the BDD method, because the methods are not always applicable to a certain type of attack tree or as effective. For example, the bottom up method is the faster method of the two. However, when an attack tree has shared subtrees the method will require a lot more work as we will show later. Then it is better to use the BDD method. There also exists a third technique, which is model deciding. This is an alternative, but more expensive and complex method. Examples of this will be given in Chapter 3.

Binary decision diagrams

A binary decision diagram is a data structure which is used to represent a Boolean function. It is a directed, acyclic graph with one root node. Every node represents a variable of the Boolean function and has two children. One for the function in case the variable equals 0 and the other for the function when the variable equals 1. There are also terminal nodes, which do not have

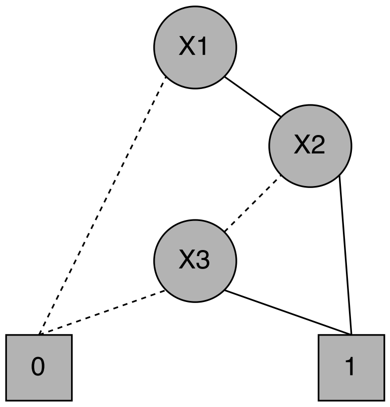


Figure 1.4: Example of a binary decision diagram of the function $f(x_1, x_2, x_3) = x_1 \cdot (x_2 + x_3)$

any children. They specify the final value of the Boolean function given a certain variable assignment. See Figure 1.4 for an example of a BDD. More about binary decision diagrams will be said in Section 2.2.

1.3 Our contribution

This thesis gives an overview of the most prominent existing methods for analysing attack trees as well as new more efficient analysing methods for attack trees. With the classical attack trees and its extensions there are four types of attack trees:

1. Classical attack trees, so no shared subtrees and SAND gates
2. Classical attack trees with the addition of SAND gates
3. Attack trees with shared subtrees but no SAND gates, so directed acyclic graphs (DAGs) without SAND gates
4. Attack trees with both shared subtrees & SAND gates, so DAGs with SAND gates

This thesis focuses on the first three types of attack trees, where we treat the first two types as one, so attack trees without shared subtrees, but possibly with SAND gates. Thus, in this thesis we will look at the analysis methods of the following two types of attack trees:

1. Attack trees with possibly SAND gates, but no shared subtrees
2. Attack trees with shared subtrees, but no SAND gates.

For the first type of attack tree, there is a method called bottom up which is already efficient to work with. We show how this bottom up method can be applied to different attributes. For the other type of attack tree this is not possible. We will show why and how the BDD method is applicable for this type of attack tree. The main idea behind the BDD method is the following. Attack trees are essentially just a Boolean function. A BDD is a data structure to represent a Boolean function. We convert attack trees to binary decision diagrams, which we find easier to analyse. This results in efficiency gains compared to the methods that are currently used. The current methods are mainly focused on model checking tools and simulations which are very complex. Current BDD methods merely focus on attack vectors, i.e. what basic attack steps form an attack, and simple risk computations. We show this can be extended to analyse more properties of attack trees in an efficient way. Table 1.1 shows the results for the analysation methods for different types of attack trees we treat in this thesis. For both types of attack trees, it shows for each attribute how it can be analysed. Attributes are the values of the attack tree, such as probability or cost. BU stands for bottom up, a method where leaf values will be propagated up the tree to the root node. This will be the method used to analyse the first type of attack trees. The second type of attack trees can be analysed by the so called BDD method, where attack trees are translated to BDDs and those will be analysed. For probability, it says BDD which means that it works with any type of BDD. Other attributes say that the method is MBDD, which means that the analysis will be done with BDDs that only show minimal cut sets.

Type	**Prob.**	**Cost**	**Time**	**Skill**	**Boolean**	**Comb.**
TREES with SAND gates	BU	BU	BU	BU	BU	Partly BU
AT with shared subtrees	BDD	MBDD	MBDD	MBDD	MBDD	MBDD

Table 1.1: Results of the thesis. It shows which attribute can be analysed in what way for both types of attack trees treated in this thesis

1.4 Thesis overview

Chapter 2 will explain the concepts of attack trees and binary decision diagrams to a greater extent. After that, Chapter 3 shows some of the related work that has already been done on this topic. Then Chapter 4 gives the overview of analysis methods for the two types of attack trees covered in this thesis. Finally, Chapter 5 is the conclusion of this thesis followed by the acknowledgements in Chapter 6.

Chapter 2

Preliminaries

This chapter will give an overview of the concepts which are essential to know for you to understand the rest of this thesis. In Section 2.1 attack trees will be discussed, which is the security model this thesis focuses on. After that, binary decision diagrams are explained in Section 2.2 which will later on be used to help analysing attack trees.

2.1 Attack Trees

Attack trees, first described by Schneier in [15]), are a formalism to model the security of a system. They are conceptual diagrams showing how an asset or target can be attacked. They have a hierarchical structure and an intuitive representation of multi-step attack scenarios. Therefore they are very practical to use and can be used for analysing the safety of a system. This is especially important if security is a critical aspect of the system, which in these days is almost always the case. See Figure 2.1 for an example. Attack trees are useful to understand several aspects:

- what the attack goals are, i.e. what steps have to be done to complete an attack
- what attacks are likely to occur, i.e. what attacks have a higher probability to occur
- the security assumptions of a system, i.e. do many attacks rely on one specific basic attack step
- where to best spend a security budget, for example if a loot of attacks exploit one part of the system, spend your security budget on that part to make a bunch of attacks a lot harder.
- what the properties of attacks are, i.e. what is the cost of an attack, what is the cheapest attack etc.

2.1.1 Attack tree structure

The first attack trees introduced by Schneier [15] represent the attacks on a system as a tree structure. An attack tree models an attack on a system, which can be achieved in multiple ways. This goal will be refined into smaller sub attacks using gates, which defines the structure of the attack tree. The root node is the goal of the attack. This root node can be refined further into sub attacks. The direct children of a node N are the sub attacks of that node N. These child nodes are conditions which must be satisfied to make the direct parent node true. When the root node is satisfied, the attack is complete. The relation between parent and child nodes can either be conjunctive, with an AND-gate, or disjunctive, with an OR-gate. The AND-gate requires all the child nodes to be true for the parent node to be true. The OR-gate requires that only one child node needs to be true for the parent node to be true. Thus basically, "OR" nodes represent different ways to achieving the same goal, whereas "AND" nodes represent different steps in achieving a goal [15]. When a node cannot be refined any further, it has no children and is a leaf node. The leaf nodes are the actual attack steps, called basic attack steps.

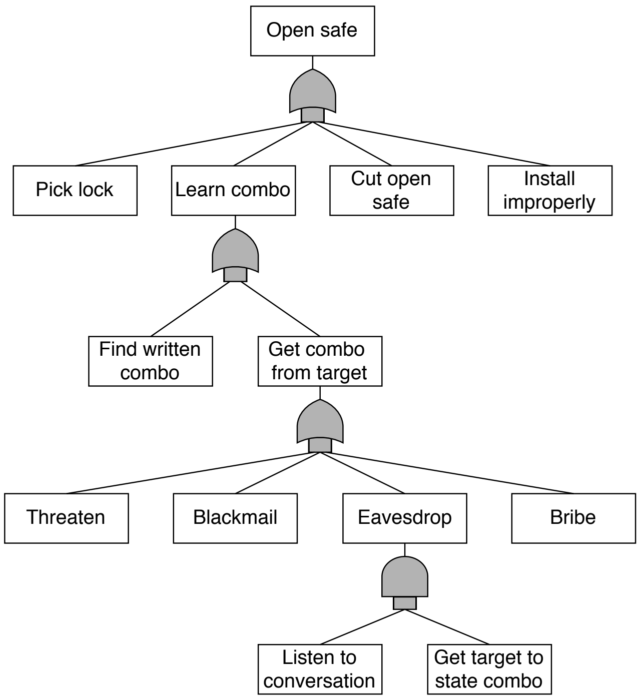


Figure 2.1: Example of a classical attack tree

Example 1. *Figure 2.1 shows an example of a classic attack tree from [15].*

The goal of this attack is to open a safe. This root node is refined by an OR gate. Opening the safe can either be done by picking the lock, learning the combo, cutting open the safe or by installing the safe improperly. Out of these 4 attacks, 3 are leaves and thus basic attack steps. Learn combo is refined further into sub attacks. To learn the combo one either needs to find the written combo somewhere or get it from someone who knows the combo. Getting the combo can also be done in several ways. One of the possibilities is eavesdropping. Eavesdropping is connected to its children via an AND-gate hence to successfully perform this sub attack, one needs to listen to a conversation of the target AND get the target to say the combo. The leaf nodes can be executed and this can be propagated up the tree. Successful attack nodes can satisfy their parent nodes according to the gates. Once the root node is satisfied, the attack is successfully executed.

Attributes

Once the attack tree is created, different values can be assigned to the leaf nodes. The set of names for these values are the *attributes* of the attack tree. Attributes are needed to analyse properties of the different attacks. Without any attributes, attack trees would only be useful to see how a system can be attacked, but it is impossible to say something about the cost, damage, time etc. of an attack. These attributes can be Boolean values, such as possible vs. impossible, or special equipment(SE) needed vs. no SE needed, but also continuous values like the cost of a basic attack step. An attack tree has some set of attributes and every leaf must contain values for all attributes, otherwise the attack tree is incomplete. If one leaf contains the cost for a basic attack step, but other leaf nodes do not, then it is still impossible to analyse the cost properties of the attack tree. Thus a leaf value can have several values, for example the cost of an attack step as well as a Boolean value like equipment needed. The information of all leaf values is called the basic attack step information, which can be stored in a table for example.

2.1.2 Attack tree definition

With all this information we can define attack trees as follows, inspired by [12].

Definition 1. *BAI: Attr $\to \mathbb{R}$ is a function that assigns values from the real numbers to attributes where Attr is the set of attributes of the attack tree.*

Definition 2. *An attack tree A is a tuple (V, Child, Top, Attr, val, L), where*

- *V is a finite set of nodes, including the set of nonterminal nodes V_N and the set of all leaf nodes V_T, where $V_N \cap V_T = \emptyset$.*
- *Child: $\mathsf{V} \to \mathsf{V}^*$ maps each node to its child nodes.*

- *Top $\in$ V is the unique top level element, representing the goal of the attacker.*
- *Attr is the set of attributes*
- *val: $V_T \rightarrow$ BAI is a function that for every BAS assigns a value to all elements in Attr using the function BAI.*
- *L: $V_N \rightarrow$ Gates is a function that assigns to each nonterminal node a gate available in the attack tree where Gates is the set of gates in the attack tree.*

We require each AT to be a directed acyclic graph with a unique root Top $\in$ V. Furthermore we define the set of edges of A by $E = \{(v, w) \mid w \in child(v)\}$ and $Leaves = \{v \in V \mid Child(v) = \epsilon\}$

As mentioned earlier, there are two extensions of attack trees which will be treated in this thesis. These are shared subtrees and SAND gates.

Attack trees with shared subtrees

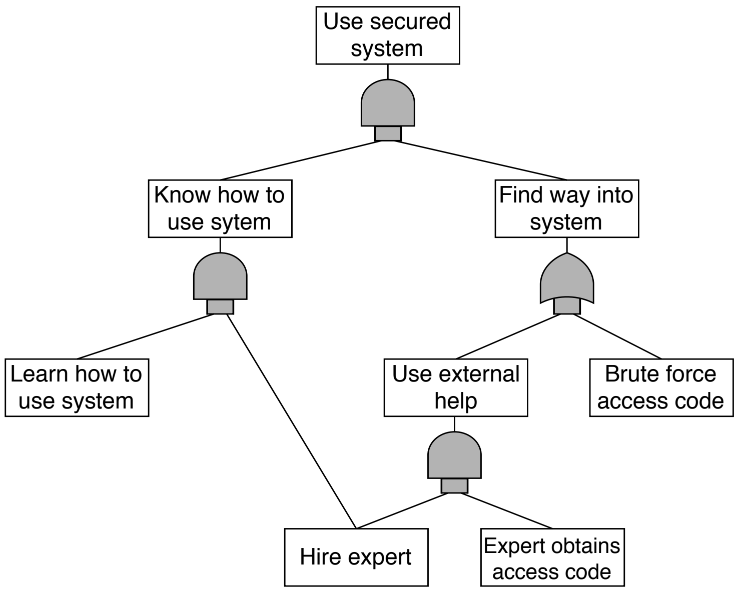


Figure 2.2: Attack tree with shared subtree

When attack trees were introduced for the first time, they had a real tree structure and could not have any shared subtrees. Every node was reachable from the root in a unique way. However, in the way we just defined attack trees, we require them to be directed acyclic graphs. This means that they do not need to have a real tree structure. This allows nodes to be reached in multiple ways from the root node. Figure 2.2 shows an example of this.

The node "Hire expert" is reachable in two different ways and therefore this attack tree does not have a tree structure, but it is a directed acyclic graph.

Attack trees with SAND gates

Finally, there is one more extension possible for attack trees which will be discussed in this thesis. Next to the standard OR- and AND-gates, there exists another gate. Temporal order is crucial in security and therefore there is a sequential version of the AND-gate, called a SAND-gate. This allows us to express the ordering of events in attack trees. It is essentially an AND-gate, but now there is an ordering on the child nodes. Not only do all child nodes need to be true for the parent node to be satisfied, they should also be performed in a certain order. An attacker will only start with the next basic attack step when all previous basic attack steps have been finished. Literature also features a SOR-gate, which is the sequential version of the OR-gate, but we will not use it in this thesis. The symbols of these gates are the same as the AND- and OR-gates, but also containing an arrow inside it indicating the order of the children, i.e. from left to right or right to left. Attack trees including a SAND gate are called *dynamic attack trees*, whereas attack trees without SAND gates are called *static attack trees.*

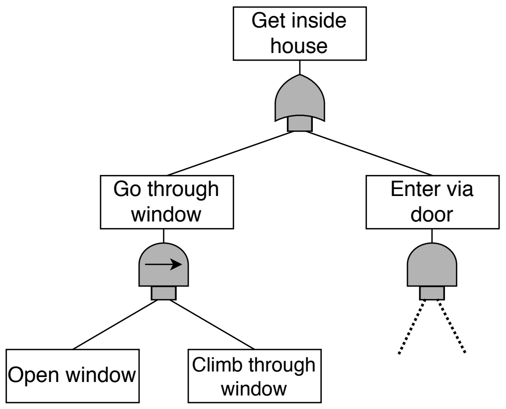


Figure 2.3: Example of a part of an AT with a SAND gate

Example 2. *An example can be found in Figure 2.3. The node "Go through window" is connected with its children via a SAND-gate. The arrow goes from left to right, so that is the order in which the child nodes have to be executed. To go through the window, first the window needs to be opened and after that is finished one can climb through the window.*

2.1.3 Attack tree semantics

When is a subset of basic attack steps an attack? This is defined by the attack tree semantics.

For static attack trees, we only care about the status vectors of attacks. It is enough to know what basic attack steps are executed and which are not since there are no operators that depend on time. The semantics of static attack trees are given by the structure function. Given the status of all basic attack steps, where 0 means not executed, and 1 means executed, the structure function indicates whether the top level attack is executed.

Definition 3. *The structure function of a static attack tree T is the function $\phi_T : \{0,1\}^n \rightarrow \{0,1\}$ that takes as input a status vector $(b_1, b_2, \dots b_n)$ of n booleans, where $b_i = 1$ if the BAS i is executed, and $b_i = 0$ otherwise [16].*

Deriving the structure function of a static attack tree is done in a top down fashion.
Formally, the structure function ϕ_T of T is defined recursively in terms of the structure functions of its children. To do, we extend the structure function with an extra parameter a that indicates the current sub attack, i.e. an intermediary node. Thus, $\phi_T(B, a)$ indicates whether attack a is executed, given the status vector B. Then $\phi_T(B)$ is defined as $\phi_T(B, TA)$ where TA is the root node.

Definition 4. *Let $T = (V, Child, Top, Attr, val, L)$ be a static attack tree. The structure function $\phi_T : \{0,1\}^n \times A \rightarrow \{0,1\}$ of T is defined as follows. We write $\boldsymbol{b} = (b_1, b_2, \dots b_n)$ for the status vectors of the basic attack steps, and $Child(a) = \{a_1, a_2, \dots a_m\}$ for the set of children of intermediate attack a [16].*

- *If $a \in V_T$, then $\phi_T(\boldsymbol{b}, a) = b_i$*
- *If $a \in V_N$ and $L(a) = AND$, then $\phi_T(\boldsymbol{b}, a) = \phi_T(\boldsymbol{b}, a_1) \wedge \phi_T(\boldsymbol{b}, a_2) \wedge \dots \phi_T(\boldsymbol{b}, a_m)$*
- *If $a \in V_N$ and $L(a) = OR$, then $\phi_T(\boldsymbol{b}, a) = \phi_T(\boldsymbol{b}, a_1) \vee \phi_T(\boldsymbol{b}, a_2) \vee \dots \phi_T(\boldsymbol{b}, a_m)$*

This does not work for attack trees including SAND gates!
An attack now is a status vector B of which the structure function will evaluate to 1, so $\phi_T(B, TA) = 1$. In an alternative view, an attack is a subtree T' such that: (1) T' contains the root node; (2) for any AND-node $v \in T'$, all child nodes of v belong to T'; (3) for any OR-node $v \in T'$, at least one child node of v belongs to T'[5].

For dynamic attack trees, however, we also need to keep the order of basic attack steps in mind. There are two interpretations of SAND gates. Take a look at Figure 2.4. Sub1 requires the basic attack steps A and B to be done in that order respectively, but sub2 requires the basic attack steps B and A to be done in the opposite order. One interpretation is very strict, which requires the basic attack steps to be done in the exact same order as defined by the SAND gate. In that case, this attack tree would be invalid. Another interpretation is that the basic attack steps have to be done in the specified

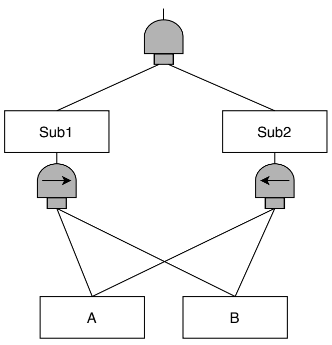


Figure 2.4: Weird AT structure

order, but if a certain basic attack step is already done in another subtree, then that is fine. In that case this structure would be perfectly fine. In this thesis, we use the first interpretation.

Because the order of the basic attack steps is also important for attack trees including SAND gates, the status vector of an attack now not only consists of the information whether a basic attack step is executed, but also the time slot when it is executed (only for basic attack steps which are executed). An input of an attack could look like this: $\{a, c\}\{f\}\{d, e\}$ These are all the basic attack steps that are executed, and a and c are executed in the first time slot, f in the second and d and e in the third time slot. First of all, for such an attack to be true, the executed basic attack steps must satisfy the root node. We can check this using the structure function given earlier. Replace every SAND gate with an AND gate, and construct the structure function of the attack tree. Now we can easily check if with the executed basic attack steps the root node is satisfied. If the input evaluates the structure function to 1, the executed basic attack steps satisfy the root node and we need to check a second part. If the structure function evaluates to 0, then this is not a valid attack. If a certain input made the structure function evaluate to 1, we need to check if the order of the sub attacks at the SAND gates are in the correct order. This can be done in a bottom up style. Every executed basic attack step has a time slot attached to it as explained above. These are propagated up the tree and during this process the checks will be done at the SAND gates. An OR node will inherit the minimum value of its children, as it is true once the first child is true. At an AND gate, the maximum value of its children will be inherited as it is only true when all children are done. The propagation at a SAND gate is the same as for an AND gate, but it will also check if its children are executed in the correct order (for example from left to right). If this is the case, the bottom up process will continue. If the basic attack steps are not done in the correct order, the algorithm will return a fail, meaning that the input is not a valid attack. If the algorithm has reached the root node without failure, true will

be returned and the input attack is valid. So an input is only a valid attack if 1) the structure function where SAND gates are replaced by AND gates evaluates to true and 2) the bottom up process returns true.
Again an alternative view, a successful attack is a subtree T' such that: (1) T' contains the root node; (2) for any AND-node $v \in T'$, all child nodes of v belong to T'; (3) for any OR-node $v \in T'$, at least one child node of v belongs to T'; (4) for any SAND-node $v \in T'$, all child nodes of v belong to T' in the same order as in the original attack tree.

Example 3. *Consider Figure 2.5 for an example. Suppose we get $\{A\}\{C\}$, then we know that both a and c are executed at time slots 1 and 2 respectively. So Sub1 is true with time value 1. This is shown in the left AT. Both child nodes of the root are true in the correct order, hence $\{A\}\{C\}$ is a valid attack. An example of an invalid attack would be $\{C\}\{B\}$, the right AT in Figure 2.5, because now Sub1 is only executed after C which is not the correct order as specified by the SAND gate of 'Root'.*

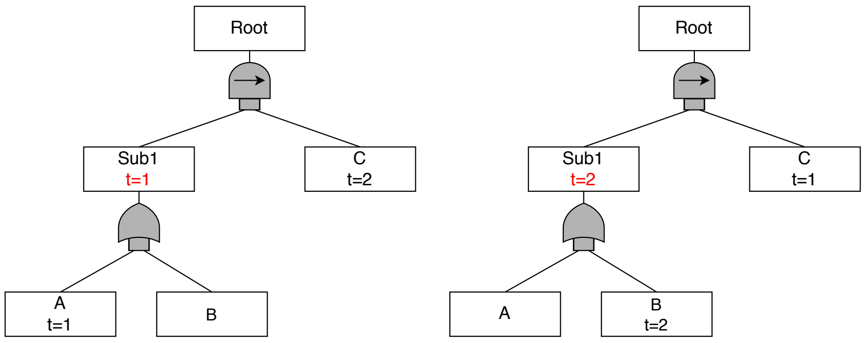


Figure 2.5: An AT structure

2.1.4 Metrics

Security metrics are the properties of an attack tree we want to calculate. Metrics we want to calculate are:

- *Probability to successfully attack.* Every basic attack step has a certain probability to be executed successfully. The probability of an attack X in the attack tree T to happen is the product of the probabilities of all basic attack steps in X that are true. X is a status vector $\{x_1, x_2, \dots x_n\}$ of basic attack steps. Then

$$P(X) = \prod_{1 \leq i \leq n, x_i = 1} P(x_i)$$

where $P(x)$ is the probability of x. We want to know what the probability is that any attack in an attack tree T is successfully executed, that is $\sum_{X \in T} P(X)$

- *Minimal cost of an attack, all attacks below a certain cost, k cheapest attacks; same metrics for time.* Every basic attack step has a certain cost to be executed. The cost of an attack X in the attack tree T to happen is the sum of the costs of all basic attack steps in X that are true. X is a status vector $\{x_1, x_2, \ldots x_n\}$ of basic attack steps. Then

  $$C(X) = \sum_{1 \leq i \leq n, x_i = 1} C(x_i)$$

  where $C(x)$ is the cost of x. We want to compute several things with this attribute, like the cheapest attack there is, X for which $\min_{X \in T} C(X)$, or the k cheapest attacks. The same holds for time.

- *Skill level needed for attacks.* Basic attack steps can require a certain skill level in order for it to be executed, for example $\{low, medium, high\}$. The required skill level of an attack X in the attack tree T is the maximum skill level of all basic attack steps in X that are true. X is a status vector $\{x_1, x_2, \ldots x_n\}$ of basic attack steps. Then the skill level required for the attack X is

  $$S(X) = \max_{1 \leq i \leq n, x_i = 1} S(x_i)$$

  where $S(x)$ is the skill level of x. We want to calculate what skill level is needed for attack and to find for example all attacks with maximum skill level medium.

- *Special equipment needed for attacks.* This could really be any boolean value. A basic attack step could either require special equipment or not. Whether an attack X requires special equipment or not, depends if there is at least one basic attack step in X that needs special equipment. We can order 'No SE' and 'SE' just like skill levels to get the same definition as with skill level. SE needed will be ordered higher than no SE needed, since SE needed dominates the final value. If there is at least one BAS which requires SE, then SE is needed for the attack. Attack X is a status vector $\{x_1, x_2, \ldots x_n\}$ of basic attack steps. Then whether SE is needed for X is specified by

  $$SE(X) = \max_{1 \leq i \leq n, x_i = 1} SE(x_i)$$

  where $SE(x)$ returns true if SE is needed and false otherwise. With this attribute we can do calculations like computing all attacks that do not require special equipment.

- *Combinations of the above metrics.* An example of this could be the cheapest attack without any special equipment.

Basically, these metrics are just examples. For example, instead of the cost of an attack, we can also take the damage an attack does. Or instead of skill levels, we could work with discrete damage levels. It is more about the different categories of metrics/attributes we treat in this thesis. Cost and time are real-valued variables, where less is better. The shorter an attack, the better it is for an attacker. Skill level and special equipment (boolean) are discrete variables, where also less is better. Probability is a special attribute on its own, because even though it does work with real numbers, it works in the range 0 to 1. Calculations with probability are different from real-valued variables such as cost.

Example 4. *Figure 2.6 shows an example of an attack tree with attribute values. In this example two attributes are shown, probability and cost. All leaf nodes contain these values, since those are the basic attack steps. However, intermediary nodes do not have these values. That is why we, for example, cannot immediately see what the cheapest attack in the attack tree is. Therefore we have certain metrics for an attack tree that we want to calculate.*

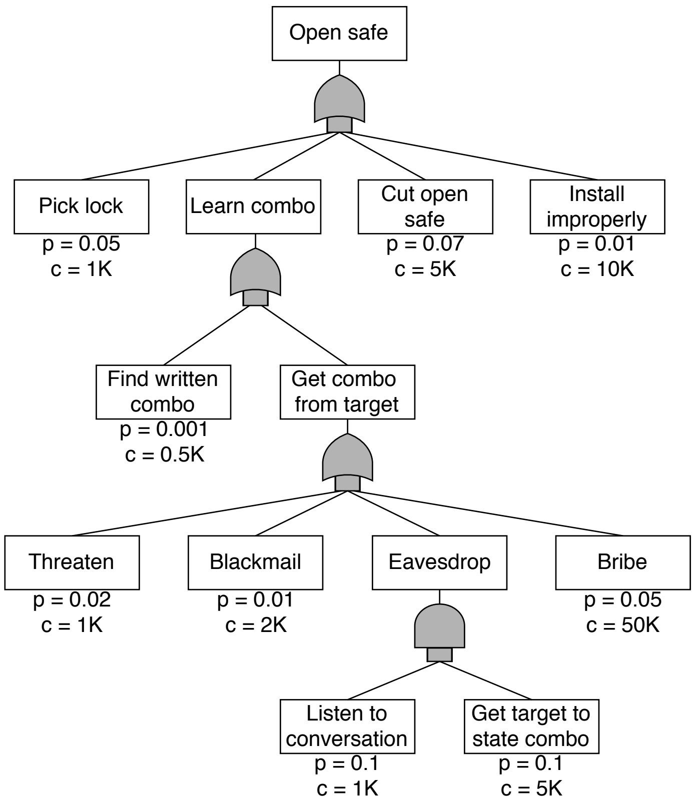


Figure 2.6: Attack tree including the attributes probability and cost

2.2 Binary decision diagrams

A key technique to analyse attack trees is by the use of binary decision diagrams. In this section we explain what BDDs are and some properties of the BDDs, such as variable ordering.

2.2.1 BDD definition

A binary decision diagram is a data structure used to represent a Boolean function. A BDD is a rooted, directed, acyclic graph. This means that it contains one root node, all edges can only be taken in one direction and there are no cycles in the graph. It has a set of nodes $V = V_N \cup V_T$ containing two types of nodes. A *nonterminal node* v which has two children $low(v), high(v) \in V$. The edge from v to its low child represents an assignment of v to 0. An edge from v to its high child represents an assignment of v to 1. It is also equipped with a variable from the Boolean function it is representing. A *terminal node* v has an attribute $value(v) \in \{0, 1\}$.

A BDD can be denoted as a tuple as follows.

Definition 5. *Let Vars be a set of variables, equipped with an order <. Then a BDD G is a tuple (V, low, high, Top, L) over (Vars, <), where*

- *V denotes the set of nodes, including the set of nonterminal nodes V_N and the set of terminal nodes V_T for which holds $V_N \cap V_T = \emptyset$.*
- *low: $V_N \to \mathsf{V}$ maps a node to its low child.*
- *high: $V_N \to \mathsf{V}$ maps a node to its high child.*
- *$\mathsf{Top} \in \mathsf{V}$ denotes the root node of the BDD.*
- *L: $\mathsf{V} \to \mathsf{Vars} \cup \{0,\ 1\}$ is a labelling function that maps each node to a BDD element. If $x \in V_T$, then $L(x) = 0 \vee 1$. If $x \in V_N$, then $L(x) = Var \in Vars$.*

A BDD is ordered if every variable is encountered in the same order for every path from the root. If this is the case then it is called an ordered binary decision diagram (OBDD).

When talking about a BDD in this thesis, we will always assume that the BDD is at least ordered. Many operations on a BDD only work if the BDD is ordered, so it is important that this will always be the case.

The labelling function assigns to each nonterminal node a variable $\in \mathsf{Vars}$ and to each terminal node a Boolean value. So $L(v) \in Vars$ if and only if $v \in V_N$ and $L(u) \in \{0, 1\}$ if and only if $u \in V_T$. Furthermore, we require the BDD to be a directed acyclic graph with only one root node.

Example 5. *Figure 2.7 shows an example of a very small binary decision diagram. This BDD represents the function $f(x_1, x_2, x_3) = x_1 \cdot (x_2 + x_3)$.*

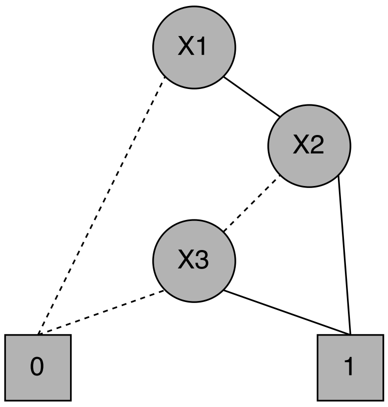


Figure 2.7: BDD of the function $f(x_1, x_2, x_3) = x_1 \cdot (x_2 + x_3)$

The circles are the different variables of the Boolean function. The two squares are the terminal nodes. A dashed line means an assignment of 0 (or false) to a variable and a solid line means an assignment of 1 (or true) to a variable.

As denoted by Bryant [4], a BDD having root node v denotes a function f_v which is defined recursively. This means that every node u has a function f_u, which is dependent on the value of node u itself as well as the values of the functions of both its children. More formally, a BDD with root node v denotes a function f_v, assuming Vars $= \{x_1, \ldots, x_n\}$ and $x_i < x_{i+1}$, defined recursively as:

- If v is a terminal node, then
  - If $L(v) = 1$, then $f_v = 1$
  - If $L(v) = 0$, then $f_v = 0$
- If v is a nonterminal node with $L(v) = i$, then f_v is the function $f_v(x_1, \ldots, x_n) = \overline{x_i} \cdot f_{low(v)}(x_1, \ldots, x_n) + x_i \cdot f_{high(v)}(x_1, \ldots, x_n)$

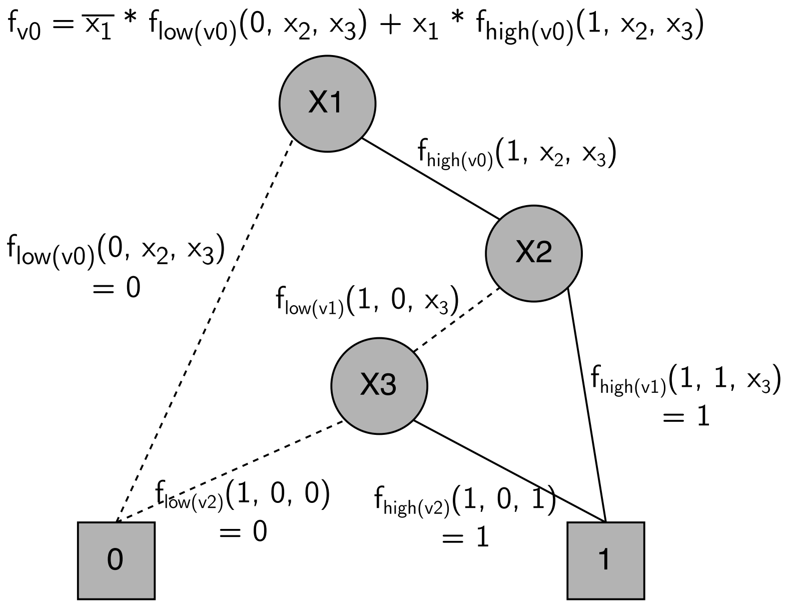
This definition is based on Shannon expansion [4][3][1]:

$$F = x_i \cdot F_{x_i} + \overline{x_i} \cdot F_{\overline{x_i}}$$

where:

- F is a Boolean function
- x_i is a variable
- $\overline{x_i}$ is the negation of x_i
- F_{x_i} and $F_{\overline{x_i}}$ are F where $F_{x_i} = F(x_1, \ldots, 1, x_{i+1}, \ldots, x_n)$ and $F_{\overline{x_i}} = F(x_1, \ldots, 0, x_{i+1}, \ldots, x_n)$

Consider the example from Figure 2.7 again of which we will derive the Boolean function using the above formula. Figure 2.8 shows the same BDD, but now with the derivation along the edges. We start at the root node,



$f_{v0} = \overline{x_1} * f_{low(v0)}(0, x_2, x_3) + x_1 * f_{high(v0)}(1, x_2, x_3)$
X1
$f_{high(v0)}(1, x_2, x_3)$
X2
$f_{low(v0)}(0, x_2, x_3) = 0$
$f_{low(v1)}(1, 0, x_3)$
X3
$f_{high(v1)}(1, 1, x_3) = 1$
$f_{low(v2)}(1, 0, 0) = 0$
$f_{high(v2)}(1, 0, 1) = 1$
0
1

Figure 2.8: BDD of $x_1 \cdot (x_2 + x_3)$ with derivation

which we call v_0. The variable corresponding to this node is x_1. v_0 is a nonterminal node hence the second case needs to be applied. As can be seen in Figure 2.8:

$$f_{v_0} = \overline{x_1} \cdot f_{low(v_0)}(0, x_2, x_3) + x_1 \cdot f_{high(v_0)}(1, x_2, x_3) \tag{1}$$

As shown in the Figure, the low edge leads to the 0 terminal node. The high edge is $f_{high(v_0)}(1, x_2, x_3)$, which equals

$$\overline{x_2} \cdot f_{low(v_1)}(1, 0, x_3) + x_2 \cdot f_{high(v_1)}(1, 1, x_3) \tag{2}$$

This equation is again split up along the outgoing edges of x_2. This process is continued until the terminal nodes are reached in every path. This is shown in Figure 2.8 with the values along the edges.
We can see that $f_{low(v_1)}(1, 0, x_3) = x_3$. Substitution into (2) gives

$$f_{high(v_0)}(1, x_2, x_3) = \overline{x_2} \cdot x_3 + x_2$$

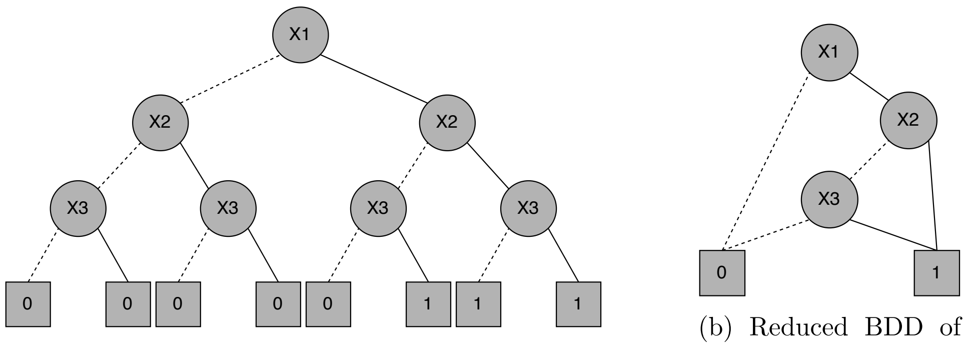
Substituting this equation and the low edge of x_1 into the top formula gives us a Boolean function

$$f_{v_0} = x_1 \cdot (\overline{x_2} \cdot x_3 + x_2)$$

which is equivalent to the Boolean function $x_1 \cdot (x_2 + x_3)$ we saw earlier. Thus a set of argument values $\{x_1, \ldots, x_n\}$ describes a path in the BDD starting from the root node ending in a terminal node. If some node v along the path has $L(v) = x_i$, then the path continues to the low child if $x_i = 0$ and to the high child if $x_i = 1$. The value of the terminal node will determine the value of the function given these arguments. Every path defined by a set of arguments is unique and every node in the BDD occurs in at least one path, so every part of the BDD is reachable.

2.2.2 BDD reduction

A BDD is reduced if it 1) contains no node v with $low(v) = high(v)$, 2) there are only two terminal nodes, one with value 0 and the other with value 1 and 3) the BDD does not contain two different nodes v and v' such that the subgraphs rooted by v and v' are isomorphic [1][4][17]. Two Boolean expressions are equivalent if and only if their reduced BDD's are isomorphic for some variable ordering.



(a) Binary decision tree for the function f

(b) Reduced BDD of the function f

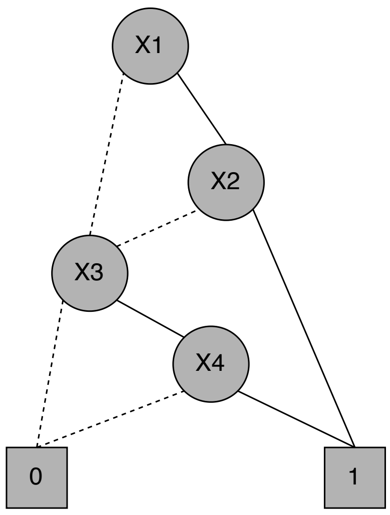
Figure 2.9: Example of BDD reduction

Example 6. *Figure 2.9 shows an example of BDD reduction. Both Figure 2.9a and Figure 2.9b show a BDD of the Boolean function*

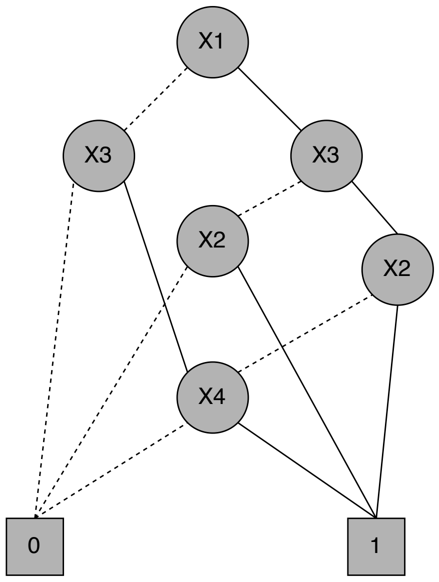
$$f(x_1, x_2, x_3) = x_1 \cdot (x_2 + x_3)$$

The BDD in Figure 2.9a models all Boolean combinations for the function f. As you can see, the BDD has more than two terminal nodes and some nodes have two children with the exact same values. Therefore this is not a reduced BDD. It is ordered because the variables occur in the same order in every possible path. It is easy to see what the truth value of the function f is given a variable assignment. Assume one wants to know the value of $f(0, 1, 0)$ then you traverse the BDD according to the variable assignment and you end up in a 0 terminal node. This means that the function f is false for the variable assignment $x_1 = 0, x_2 = 1, x_3 = 0$.

2.2.3 The impact of variable ordering



(a) BDD with good variable ordering for $x_1 \cdot x_2 + x_3 \cdot x_4$, namely $x_1 < x_2 < x_3 < x_4$



(b) BDD with bad variable ordering $x_1 \cdot x_2 + x_3 \cdot x_4$, namely $x_1 < x_3 < x_2 < x_4$

Figure 2.10: Impact of variable ordering on BDD size

The size of the BDD heavily depends on the ordering of the variables. For example, Figure 2.10 shows two BDDs, both of the function

$$g(x_1, x_2, x_3, x_4) = x_1 \cdot x_2 + x_3 \cdot x_4$$

Figure 2.10a uses a good variable ordering $x_1 < x_2 < x_3 < x_4$, while Figure 2.10b uses a worse variable ordering $x_1 < x_3 < x_2 < x_4$. This example clearly shows the impact of the variable ordering on the size of the BDD and thus how important it is to work with a good ordering. Computing an ordering that minimizes the size of the graph is a co NP-complete problem. However, experience has been that with some understanding of the problem domain an appropriate ordering can be found fairly easy [4]. Furthermore there are efficient heuristics which can find good orderings.

Chapter 3

Related Work on Attack Tree Analysis

Below, we discuss two categories of related work: attack tree analysis with adaptation of attack tree semantics and with the help of external tools.

3.1 Semantics

The first category is focussed on adapting the attack tree semantics to better analyse them. The authors of [13] argued that to be able to answer complex questions about attack trees, it is necessary to provide attack trees with foundations. In [13] attack trees are provided with a semantics in terms of attack suites and define valuations and projections in a formal way. An attack tree simply defines a collection of possible attacks which they call an *attack suite*. Each attack consists of the components required to performed an attack. These attack components are at the lowest level of abstraction and thus have no internal structure. Using these semantics information on the interpretation and grouping of attacks or any causal relations between attacks is discarded, like being ordered in time. The difference in structuring can arise from a different approach towards partitioning the attacks. Therefore two attack trees with a different structure may be equivalent. This problem does not occur with their representation. Furthermore they show what attributes can be analysed using this representation and which ones do not work. Their main result is a formalization of the concepts informally introduced by Schneier as we have seen earlier in Section 2.1. This formalization clarifies which manipulations of attack trees are allowed under which circumstances. This understanding is very important for building adequate tool support. Another paper, [11] describes the forest of the trees, which is a good overview of the different security models used to analyse the security of a system.

3.2 Analysis methods

Other papers focus on using external tools or creating external tools to help them analyse attack trees. In [7], attack trees are analysed by converting them to Petri nets. Then Monte Carlo simulation is used to analyse the Petri nets. Questions like average time for successful attack, minimal cost, maximal effort can be posed. So in this paper it is shown how to build a petri net from an attack tree, how to analyse it using simulation and how the results can be used to further refine the attack tree or to develop countermeasures. In [2] the analysis of a slight variant of attack trees using BDDs is shown. Weighted attack defense trees (WADT) are analysed using multi terminal binary decision diagrams (MTBDD), an extension of binary decision diagrams that allows a more general and efficient evaluation tool for the weight functions associated to a WADT. An attack defense tree (ADT) is an extension of an attack tree where it is allowed to also model countermeasures, see [10] and [11]. A method is shown to efficiently compute various quantitative and probabilistic measures. This is done by translating the WADT to a MTBDD which allows the modeler to evaluate the probability distribution function of the cost and impact related to any possible attack scenario. In the paper the authors want to enlarge the view of WADT analysis by evaluating the distribution of cost and impact, i.e. to find which is the probability of reaching a successful attack given a cost and impact. The paper [12] describes the analysis of attack trees with a model checker Uppaal CORA. With this model more complex gates, temporal dependencies between attack steps, shared subtrees and realistic, multi-parametric cost structures can be handled. To analyse an attack tree, they have provided a compositional semantics in terms of priced timed automata (PTA). Each AT element is translated into a PTA and the PTA of the entire attack tree is analysed by formulating the security measures as queries in the logic mWCTL. [16] is about fault tree analysis. Fault trees are the same as attack trees, except that they model faults in a systems instead of attacks on a system. One of the described analysis methods is the BDD-method. Here fault trees are translated to BDDs which can then be used to analyse the properties of the fault tries. The basic elements of a fault tree normally contain probability, which is the chance that elements in a system fails. By propagating these values up the tree the failure probability of the whole system can be calculated. In [16] they also show that this is not easy to do when the fault tree has shared subtrees, because then one needs bayes law to calculate dependent probabilities. Also they show that this can be avoided with BDDs and thus makes the process a lot easier. However, this is only shown for fault trees and also only for probabilities. We will show that this can also be used for other properties in attack trees such as cost and time.

3.3 Application

[9] shows a case study of how attack-defense trees are used to analyse threats and countermeasures in an ATM. This is a great example of how such a model, in this case the attack-defense trees, can be applied to a real life situation to analyse threats.

Chapter 4

Analysis of attack trees

This chapter surveys two analysis methods for different classes of attack trees.

4.1 Overview

4.1.1 Types of attack trees

Recall the two different attack trees which will be discussed, also see Table 4.1. It is important to work with different kind of attack trees, because they all need to be analysed in a different way. The two types are:

- **Attack trees with possibly SAND gates**
  This type of attack tree contains one root node, it has three gates, AND-, OR- and SAND-gates and there are no shared subtrees, i.e. a node can only be reached in one unique way starting from the root node. Therefore this form of attack tree does have a real tree structure. It will be analysed with the bottom up method, where leaf values will be propagated up the tree to find the property of the root node of the attack tree.

- **Attack trees with shared subtrees and no SAND gates**
  This type of attack tree contains shared subtrees, but there is no SAND-gate. This means that a certain node in an attack tree can have two or more parents thus there is no unique way from the root node to that node anymore. Therefore this form of attack tree does not have a real tree structure, but it is a directed acyclic graph (DAG). This type of attack tree will be analysed with BDDs, where the attack tree is first converted to a BDD and then the BDD is analysed.

Even though AND-gates are well known, for the attribute time the AND gate can have two interpretations: either it has x steps to do and those can be done in any order but not simultaneously or it has x steps to do which can be done parallel. This heavily depends on the situation, for example the group which is attacking a system. If there is just one attacker then the attacker has to do everything by himself, so it is not possible to do all steps at once. However if a big group of attackers is attacking a system they can divide all the tasks over different people hence the attack can be done faster. When analysing time, this will be taken into consideration.

4.1.2 Results

Table 4.1 shows the results for the analysation methods for different types of attack trees. For both types of attack trees we discuss in this thesis, it shows for each attribute how it can be analysed. BU stands for bottom up, a method where leaf values will be propagated up the tree to the root node. This will be the method used to analyse the first type of attack trees. The second type of attack trees can be analysed by the so called BDD method, where attack trees are translated to BDDs and those will be analysed. For probability, it says BDD which means that it works with any type of BDD. Other attributes say that the method is MBDD, which means that the analysis will be done with BDDs that only show minimal cut sets.

Type	**Prob.**	**Cost**	**Time**	**Skill**	**Boolean**	**Comb.**
TREES with SAND gates	BU	BU	BU	BU	BU	Partly BU
AT with shared subtrees	BDD	MBDD	MBDD	MBDD	MBDD	MBDD

Table 4.1: Results of the thesis. It shows which attribute can be analysed in what way for both types of attack trees treated in this thesis

4.2 TREES with SAND gates

This section will be concerned with attack trees that can include SAND gates, but no shared subtrees. This type of attack tree can be analysed by the so called bottom up method. All basic attack steps of the attack tree have attribute values. These values need to be propagated up the tree to the root node to say something about the whole attack tree, for example the cheapest attack. A node's value is a function of its children's. There are different calculation rules for AND, SAND and OR nodes [15]. In summary, we start at the leaf nodes and using the different calculation rules we calculate up to the root.

Probability

Suppose every basic attack step has a certain probability to be executed successfully, which are all independent of each other. Recall that the probability for a certain attack to happen is the product of the probabilities of all basic attack steps in the attack. Using the bottom up technique the total probability that any attack will be successfully executed can be found. For this the basic probability rules can be used. Suppose you have an attack tree with the probabilities of all basic attack steps, which are all independent. Using the basic probability rules it is now possible to propagate these up to the root [16]:

- OR node: the addition rule, that is $P(A\ or\ B) = P(A) + P(B) - P(A\ and\ B)$.
- (S)AND node: the multiplication rule, which is $P(A\ and\ B) = P(A) \cdot P(B)$ since the probabilities for all basic attack steps are independent. The SAND nodes and AND nodes can be treated in the same way since the order of the events does not change the probability calculation.

After propagating the values at the leaf nodes to the root, the final value at the root will be the probability that any attack in the attack tree is executed. This method works because there are no shared subtrees, so if the basic attack steps are independent, then so are the subtrees of a node.

Example 7. *Consider the attack tree in Figure 4.1, which is a smaller version of the attack tree we have seen earlier in Section 4.2. Every basic attack step has a probability to be executed successfully.*
The bottom up method works as follows. 'Eavesdrop' is an AND-node hence we use the multiplication rule. Its children have probabilities of 0.1 and 0.1, so the probability for 'Eavesdrop' to happen is $0.1 \cdot 0.1 = 0.01$*. 'Learn combo' is an OR-node with as children 'Find written combo' and 'Eavesdrop' which have probabilities 0.05 and 0.01 respectively. Using the addition rule we obtain a value of* $0.05 + 0.01 - 0.05 \cdot 0.01 = 0.0595$*. Now we have reached the*

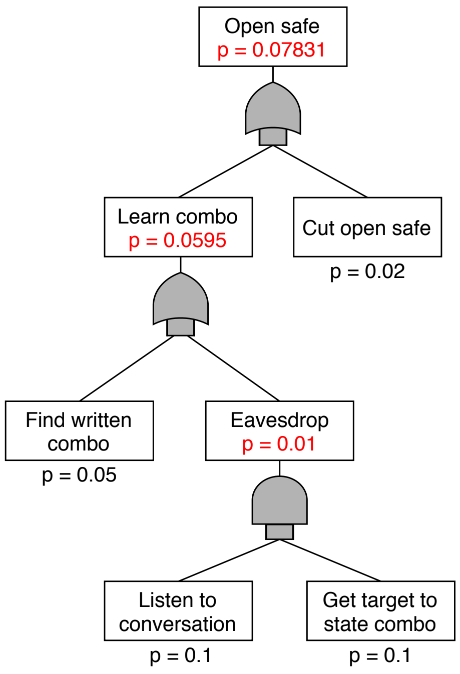


Figure 4.1: Attack tree with probabilities and the propagated values in red

root node and the probabilities of both its children are known hence we can calculate the root value. The root is an OR-node, so we get $P(Opensafe) = 0.0595 + 0.02 - 0.0595 \cdot 0.02 = 0.07831$. *So the total probability that the safe will be opened in any of these ways is 0.07831.*

Cost

Suppose every basic attack step has a certain cost to be executed. Recall that the cost of an attack will be the sum of the costs of all basic attack steps. An interesting property to find is the cost of the cheapest attack.

Cheapest attack

Finding the cheapest attack can also be achieved using the bottom up method. Only the calculation rules will differ compared to the probability attribute. These calculation rules now are [15]:

- OR node: the value of the parent will be the minimum of all child values. Let $H = \{h_1, h_2, \ldots h_n\}$ be the set of all child nodes. Then the value at the parent node will be $\min_{1 \leq i \leq n} C(h_i)$. At an OR node only one child has to be executed for the parent to be true, so that is why we take the cheapest option when we want the cheapest attack.
- (S)AND node: the value of the parent will be the sum of all child values. Let $H = \{h_1, h_2, \ldots h_n\}$ be the set of all child nodes. Then

the value at the parent node will be $\sum_{1 \leq i \leq n} C(h_i)$. At (S)AND nodes there is no choice but to execute all children for the parent to be true. Therefore the cost of a (S)AND node will be the sum of all child value. Again, the SAND nodes and AND nodes can be treated in the same way since the order of the events does not change the cost calculation.

After all leaf values are propagated upwards, the root will now contain the value of the cheapest attack. However, to be able to also know what this attack is instead of only knowing the cost of the cheapest attack, we will find the subtree which belongs to the attack. This will be done by traversing the attack tree in a certain way. Start at the root node v and apply the following in a recursive manner:

- if v is a BAS, terminate
- if v is an intermediary node:
  - If V is a (S)AND node, add all child nodes to the subtree (in the same order as in the original attack tree for SAND gates) and apply this algorithm to all children
  - If v is an OR node, add the cheapest child c to the subtree and apply this algorithm to c

This process will result in a subtree which represents the cheapest attack. This is especially useful for SAND nodes, because one can not only see what basic attack steps are part of the attack, but the subtree also shows the structure between these basic attack steps, i.e. in which order sub attacks have to be executed.

Example 8. *Figure 4.2 shows an example of how the result would look like. The costs for all basic attack steps are known at the start. These are all shown underneath the basic attack steps. Inside the boxes of the intermediary nodes are the cheapest costs to execute that sub-attack. These values are calculated using the bottom up technique as described above. Once this is finished every node should have a cost. Now from the root node we go backwards to actually find the cheapest attack. We do this in a recursive manner as described earlier. The root node is an OR node, so we add the cheapest child to the subtree which is 'Learn combo'. This is again an OR node where the cheapest child is 'Eavesdrop'. 'Eavesdrop' is an AND-node hence all children are added to the subtree. All children of 'Eavesdrop' are basic attack steps, so this traversing algorithm stops there. The subtree representing the cheapest attack is shown by the dashed edges in Figure 4.2.*

All attacks cheaper than x
If we want to use the bottom up method to find all attack cheaper than a certain cost x, then nodes will need to store more than just the cheapest

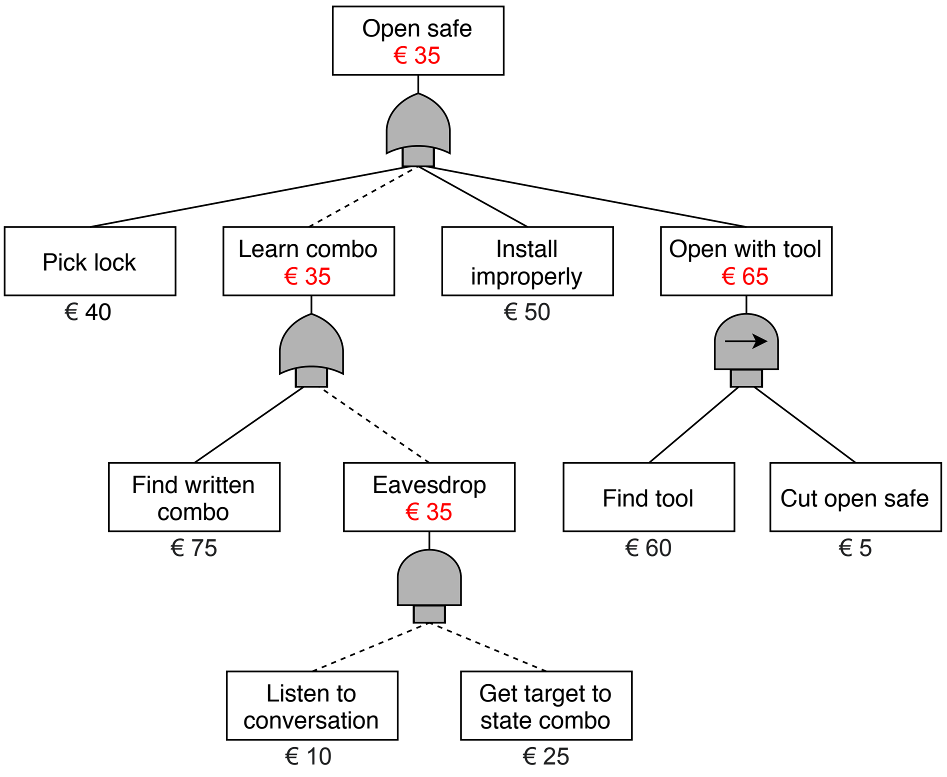


Figure 4.2: Example of finding the cheapest attack tree in a bottom up manner. The propagated values are marked in red. The dashed edges show the subtree of the cheapest attack

value. Suppose there is an OR node which has three children which can all be executed in one way, then the OR node has three ways to be executed successfully. If all of those three ways are cheaper than x, all values need to be stored. Therefore the propagation rules change for the different nodes.

- (S)AND node: the values at these nodes will be all values less than x of all possible combinations between all its child values. The new value will also need to store a child pointer to know from which child values this combination is. See an example of this below.

- OR node: the values at an OR node are all values of all child nodes. At an OR node there is a choice of how to complete this node. Therefore all possible ways of all children need to be taken into account. There is no check needed if a new value at an OR node will be higher than x, because the new values will only be inherited from the child values and the child nodes do also not contain values which are higher than x. Also this value will need to store a child pointer to know from which child value the new value originates. For example, suppose an OR node v which has two children where one child has two values and the other child has one value, then the node v will inherit all three values.

Example 9. *Suppose there is an AND node v which has two children a and b. Both a and b have two values which we will call $a1, a2, b1$ and $b2$ here. Then all possible combinations are $a1b1, a1b2, a2b1$ and $a2b2$. The new cost will be the sum of all cost values in the new combination. Such a combination can only be added to the AND node v if the new cost is less than the constraint x. Furthermore, the new value should also store from which child values this combination is called the child pointer. So the new value $a1b2$ should store as children $a1$ and $b2$. This is needed for backtracking the path of an attack. The number of possible combinations for an AND node v is given by the formula*

$$\prod_{i=0}^{nrChilds(v)} nrValues(child(v,i))$$

where $nrChilds(v)$ is the number of childs the node v has, $nrValues(v)$ is the number of cost values of node v and $child(v,i)$ returns child number i of node v.

After propagating all leaf values up, the root node will contain a certain amount of values which are all costs of all attacks cheaper than x.

The principle for backtracking to find the subtree of a certain attack stays the same. The same traversing algorithm can be used, but which child to add to the subtree now changes a bit. Instead of adding the cheapest child at an OR node, one must add the node to which the child pointer at the current node points. At (S)AND nodes still all children will be added to the subtree, but also here it is important to also choose the correct value at a child node, such that the algorithm can also determine for the child node how to continue.

Time

For analysing the time in attack trees we can distinguish two cases for AND-gates as stated earlier. When a system is attack by an individual, it is plausible to assume that he cannot execute attack steps simultaneously and thus have to perform all attack steps one after another. So in this case the AND-gate is treated as it is used in most cases. But if a system is attacked by a group of attackers who have a lot of resources and enough people to perform all attack steps at the same time, then the attack can be done much quicker. So in this case all steps of an AND-gate still have to be performed, but they can all be done at the same time. Therefore there are two cases: the non-parallel AND-gate and the parallel AND-gate.

Non-parallel AND-gate
In this case analysing time will be the same as cost. The time to perform an attack will be the sum of the time of all basic attack steps in the attack. Both are continues values which use the same rules for the bottom up technique. This will be different for the parallel AND-gate.

Parallel AND-gate
In this case, all children of an AND node can be done at the same time. Suppose an AND gate has three children with time values 4, 5 and 2. The total amount of time to execute all children will not be 11, but 5 since they can all be done simultaneously. This requires slight changes in the propagation rules for the bottom up method.
For finding the cheapest attack, the rules at the OR and SAND gate stay the same. Note that a SAND gate can never be done in parallel, because it requires a certain order on the events. The rule for the AND gate used to be to sum all time values of its children, which will be changed to be the maximum of the time values of its children. So we obtain the following propagation rules:

- OR node: the value of the parent will be the minimum of all child values. Let $H = \{h_1, h_2, \dots h_n\}$ be the set of all child nodes. Then the value at the parent node will be $\min_{1 \leq i \leq n} T(h_i)$ where $T(h_i)$ returns the time it takes to sub attack h_i.

- AND node: the value of the AND node will be the maximum of all child values. Let $H = \{h_1, h_2, \dots h_n\}$ be the set of all child nodes. Then the value at the parent node will be $\max_{1 \leq i \leq n} T(h_i)$ where $T(h_i)$ returns the time it takes to sub attack h_i.

- SAND node: the value of a SAND node will be the sum of all child values. Let $H = \{h_1, h_2, \dots h_n\}$ be the set of all child nodes. Then the value at the parent node will be $\sum_{1 \leq i \leq n} T(h_i)$ where $T(h_i)$ returns the time it takes to sub attack h_i.

Traversing the attack tree from the root to find the actual cheapest attack stays the same, because all children of an AND node still need to be completed for the AND node to be true.

For finding all attack shorter than a certain time x, also only the AND gate rule will change. The new values at an AND node will still be all possible combinations of its child values, but the new time value will not be the sum but the maximum of these child values. Note that if we take the maximum, it is impossible for the new time value to be higher than at any child, so every possible combination is for sure shorter than x. If not, then something went wrong at a child because value which are $\geq x$ should never be stored. Finding the subtree belonging to a time at the root node still works the same way by traversing the attack tree.

Skill

Suppose every basic attack step has a certain skill level needed to be executed. There are several skill levels which can be ordered, for example low, medium and high. Attackers with a low skill level will not be able to execute basic attack steps of medium or high skill. Recall that the skill level required for an attack will be the maximum skill level among all basic attack steps in the attack. An interesting property to look at is the subtree which shows all attack with a skill level below a given skill level, e.g. all attacks with a skill level lower than medium.

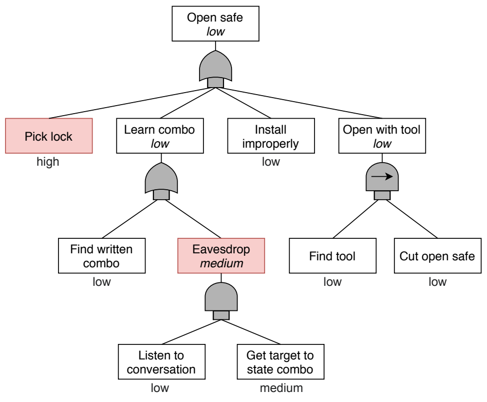
All attacks below a certain skill level

For this we will first use the bottom up method to find the minimum skill level of every intermediary node and then we will remove all nodes which have a skill level that is too high. For the bottom up method, the following calculation rules apply.

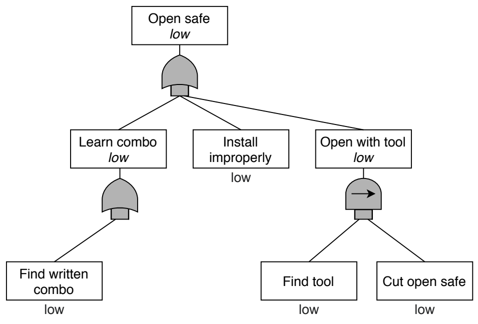
- OR node: the minimum skill level of its children is propagated up, because only one child needs to be true and we want to find the minimum skill level. Let $H = \{h_1, h_2, \dots h_n\}$ be the set of all child nodes. Then the value at the parent node will be $\min_{1 \leq i \leq n} S(h_i)$.
- (S)AND node: The maximum skill level of all its children will be propagated up, because all children need to be true hence the skill level will be the maximum of all its children. Let $H = \{h_1, h_2, \dots h_n\}$ be the set of all child nodes. Then the value at the parent node will be $\max_{1 \leq i \leq n} S(h_i)$.

If propagating all values up to the root of the attack tree is done every node in the attack tree will have its minimum skill level. Note that if the root node has a value of medium, then there is no attack with a skill level of low. The nodes with a skill level that is too high which can be reached from the root node without crossing another node with a skill level that is too high need to be deleted, but also their descendants need to be deleted. We will illustrate this with an example.

Example 10. *Figure 4.3a shows an attack tree on which the bottom up method for skill levels has been applied. The set of possible skill levels is $\{low, medium, high\}$. We want to find all attacks with a skill level lower than medium. Therefore after the bottom up all nodes, and their descendants, which have a skill level that is too high and are reachable from the root without crossing another node with a skill level that is too high need to be deleted. These nodes are marked in red. After deleting those nodes and their descendants we obtain the attack tree in Figure 4.3b, which now only shows attacks with a skill level lower than medium. Note that we can just delete all nodes which are too high and reachable from the top, because their parents do not need them for an attack. If this would have been the case*



(a) AT with all propagated skill levels. Subtrees with red marked nodes as root have to be deleted



(b) Resulting AT

Figure 4.3: Example of bottom up method for skill

then their parents would have had a skill level which was also too high and then their parent would have been deleted. So we can assure that the result attack tree only contains valid attacks.

All attacks above a certain skill level
The task of finding all attacks above a certain skill level is a lot harder. This is because you cannot just discard nodes with a too low skill level, because later on in the attack tree a node with a low skill level might come together with a node of a high skill level which would make the node with a low skill level a valid part of the attack. We did not have the problem at the previous algorithm, because higher skill level dominate at an AND node. Take a look at Figure 4.4. Suppose we want to find all attacks with a skill level of at least medium. Out of the four possible attacks ac, ad, bc and bd three of them satisfy this condition. However, ac does not because that attack requires a skill level of low. It is not possible to delete either a or c, because then ad or bc would also not be part of this attack tree anymore. This shows how hard this situation is.
So when trying to solve this with the bottom up method, we cannot just discard nodes with a low skill level, but we need to store them all at the parents to find all possible combinations of basic attack steps. This now basically comes down to finding every possible attack there is and just looking at its skill level, which is not a fast way to do it. We did not find a faster method to find all attacks above a certain skill level.

Boolean

Every basic attack step can contain a boolean value, for example whether special equipment is needed or not or if something is impossible / possible to

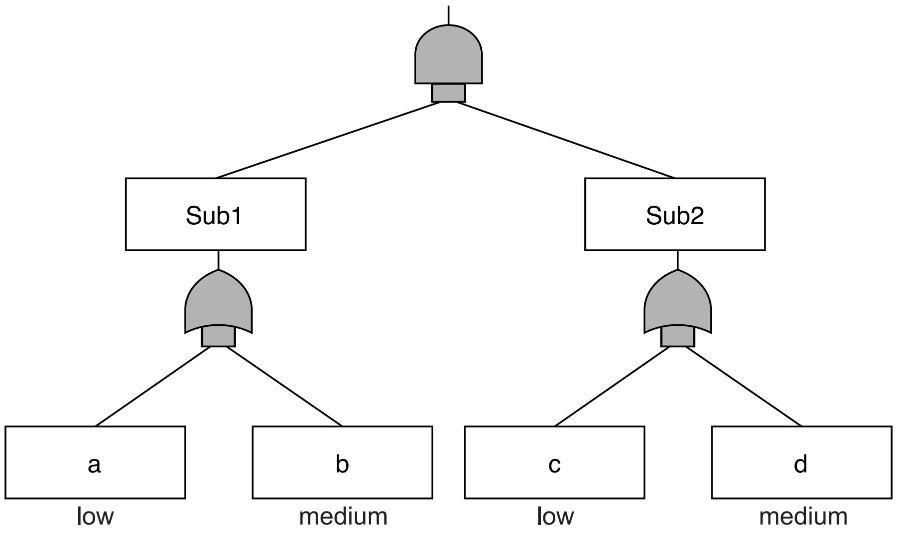


Figure 4.4: An AT structure with skill levels

do. Calculating values for this works the same as with skill level, only now there are just two options true and false whereas with skill level there could be more. True and false can also be ordered just like skill levels, without false being lower than true. For example, an attack can only be done without special equipment if all basic attack steps can be done without. As soon as there is one basic attack steps which requires special equipment an attack needs special equipment. To compare this with skill levels, an attack is only of skill level low if it only contains basic attack steps with skill level low, but if there is one basic attack step with medium, the attack will have a skill level of medium. So the way of analysing boolean values is the same as analysing skill level as described above.

Combinations

Some combinations of attributes are also possible using the bottom up method. Often you want to optimize one attribute and put constraints on the other attributes. For example the cheapest attack of all low skilled attacks. This is possible by first finding the subtree of all low skilled attacks. Then on the subtree one can apply the bottom up method to find the cheapest attack of that subtree. This is not possible for finding the cheapest attack of all attacks faster than a given time, because the bottom up method for finding all attacks faster than a given time does not return a subtree. If the attack tree does not contain a SAND gate, this can be done using the BDD method as will be described Section 4.3. If the attack tree does contain SAND gates, then other tools have to be used which are able to analyse attack trees with SAND gates, for example [12].

4.3 Attack trees with shared subtrees

This section will be concerned with static attack trees with shared subtrees. This type of attack tree can be analysed by the so called BDD method. We show how this works by first translating an attack tree to a BDD and then providing algorithms to analyse the BDDs.

4.3.1 Why BDD method?

But first we will explain why this type of attack tree is not suitable for the bottom up method. Recall that the top node is the ultimate goal of an attack and that the leaves, the basic attack steps, contain the values for the attributes. We want to derive the attribute value for the top node, but only the values for the leaves are known. Therefore, most of the time, these values need to be propagated upwards to find the value of the root node. For attack trees with a real tree structure, which we have seen in the previous section, a bottom up method can be used.
Shared subtrees, however, can lead to some difficulties when using these attack trees in computations. In [16], it is shown that for fault trees, which are closely related to attack trees, the bottom up method gets significantly harder when there are shared subtrees. This is also the case in Figure 4.5. Suppose that every basic attack step has a certain probability to be executed successfully. Then the values can be propagated upwards to the nodes "Know how to use system" and "Find way into system" using the standard probability laws. For the root node this not work anymore, because both subtrees of the root node contain the basic attack step "Hire expert". For this Bayes law needs to be used

$$P[A] = P[A|B] \cdot P[B] + P[A|\neg B] \cdot P[\neg B]$$

where $P[x]$ here means the probability that event x will fail and $P[x|y]$ the probability that x will fail given that y fails.
The intuition behind this is as follows. If there are two events and one is dependent on the other this needs to be taken into account. Assume there are two variables A and B, where A is dependent on B. To calculate the probability that A is true we need to distinguish two cases: B is true and B is false. The probability that A is true given that B is true is normally known. The same holds for the probability that A is true given that B is false. Now those probabilities should be added together to get the probability that A is true. However, one thing has to be noted. We can only use the probability that A is true given B is true if B is actually true, so we need to multiply it with the probability that B is true. The same goes for the case if B is not true. Therefore we get that the probability that A is true is equal to the probability that A is true given B is true times the probability that B is actually true plus the probability that A is true given B is false times the

probability that B is actually false.
Stoelinga and Ruijters show that the computation is easy with no shared leaves or subtrees, but it requires more complex computations if Bayes law is needed. Therefore they introduce the BDD method. Stoelinga and Ruijters observe the following with this method:

- Every path in the BDD leading to a 1-leaf corresponds to a set of status vectors, i.e. an assignment to all basic events, that make the FT fail.
- The probability for that set of status vectors to occur is obtained by multiplying their probabilities.
- Every path in the BDD corresponds to a disjoint set of status vectors. Hence, their probabilities can be added.

This method always works, irrespective of the chosen BDD ordering. This is indeed correct for probabilities. However, attack trees can also have other leaf attributes than probability. For some of these attributes the chosen BDD ordering does influence the result. We will mention this when this is the case and show how to make sure the BDD method can still work.

So [16] showed how the BDD method works for fault trees using probabilities. In this section we show how to use the same method for attack trees, where we also take care of other attributes than probability.

4.3.2 Translating AT to BDD

Binary decision diagrams represent Boolean functions hence we can express attack trees with binary decision diagrams. The Boolean function of a static attack tree, i.e. an attack tree without SAND gates, is given by its structure function, as described in Section 2.1.3.

Figure 4.5 displays the attack tree with shared subtrees from Section 2.1. We will derive the structure function for that attack tree in a top-down fashion. We start with the root, Use secured system. This is an intermediate node hence it has a gate attached to it. Now the root node should be split up in all child nodes with AND-gates between them. This process is repeated until there are only leaf nodes left. We obtain the following structure function for this attack tree

$$(\textsf{Learn how to use system} \wedge \textsf{Hire expert}) \wedge$$

$$((\textsf{Hire expert} \wedge \textsf{Expert obtains access code}) \vee \textsf{Brute force access code})$$

Once we have the structure function of a static attack tree, we need to convert this to a BDD. For this we need the notion of cut sets and minimal cut sets, as described in [16].

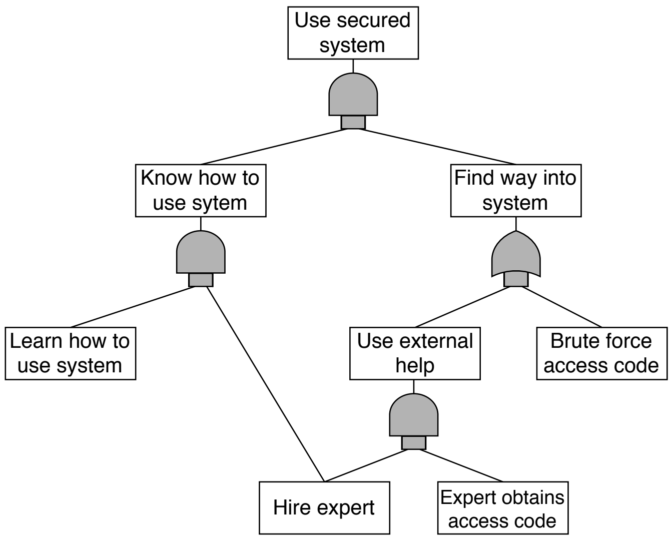


Figure 4.5: Attack tree with shared subtree

Definition 6. *A cut set is a set of basic attack steps that together cause the root node of the attack tree to be satisfied, meaning that the attack is completed successfully. In other words, a cut set lets the structure function evaluate to true. A minimal cut set is a cut set if no basic attack step can be left out of it.*

First we will describe how to convert the Boolean function to any BDD. After that we will describe how to construct a BDD with only minimal cut sets (MBDD).

We need to assume an order on the variables, because we only work with ordered BDD's. As was already shown in Section 2.2.3, the variable ordering has a large impact on the BDD size, and therefore on the efficiency of the analysis methods. With some domain knowledge and heuristics a fairly good variable ordering can be found. To construct a BDD from a Boolean formula, Shannon expansion is used to construct the top node v_0 of the BDD.

$$f_{v_0}(x_1, x_2, \ldots, x_n) = \overline{x_1} \cdot f_{low(v_0)}(0, x_2, \ldots, x_n) + x_1 \cdot f_{high(v_0)}(1, x_2, \ldots, x_n)$$

and then apply the Shannon expansion to the children $f_{low(v_0)}(0, x_2, \ldots, x_n)$ and $f_{high(v_0)}(1, x_2, \ldots, x_n)$. Recursively applying this expansion until all variables have been converted into BDD nodes yields a complete BDD [16]. However, it is not yet reduced. To accomplish this one can simply use Bryant's function Reduce [4].

Some attributes of an attack tree require the BDD method to work with BDD's that only contain minimal cut sets. This is because an attack in practice only really is an attack if the cut set is minimal. When you for example want to find the cheapest attack, then the search space only needs to be the set of minimal cut sets. Therefore this needs to be taken into account when converting the AT to a BDD. A status vector $x_1, x_2, \ldots, x_n$ represents a minimal cut set in two cases:

1. if $x_1 = 0$, then $x_2, \ldots, x_n$ must represent a minimal cut set.
2. if $x_1 = 1$, then $x_2, \ldots, x_n$ must represent a minimal cut set, and furthermore, $0, x_2, \ldots, x_n$ must not be a cut set. If $0, x_2, \ldots, x_n$ were to be a cut set, then $1, x_2, \ldots, x_n$ can never be a cut set since x_1 is superfluous.

The above cases are expressed in the following formula [16]:

$$f_{min}(x_1, x_2, \ldots, x_n) = \overline{x_1} \cdot f_{min}(0, x_2, \ldots, x_n) +$$

$$x_1 \cdot f_{min}(1, x_2, \ldots, x_n) \cdot \neg f(0, x_2, \ldots, x_n)$$

To compute the BDD with only minimal cut sets, Shannon expansion can be used for f_{min} with the same process as described earlier. To make sure the BDD is reduced, Bryant's function Reduce [4] can be used.

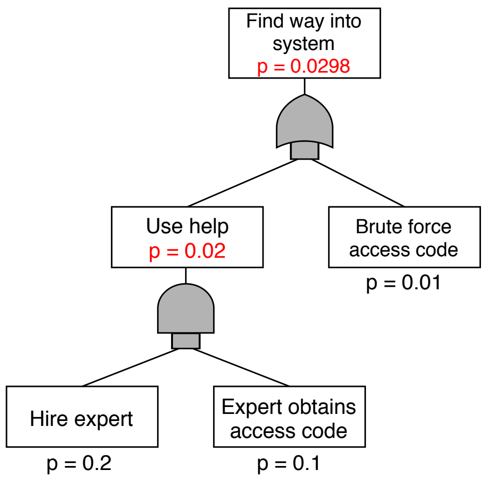
In the following section several algorithms will be described which will run on a BDD. In some of these algorithms several nodes and/or edges may be deleted. Officially we can not speak of a BDD then anymore, but rather a partial BDD. However, the goal of this method is not to keep working with BDD, but to use them in algorithms to be able to analyse attack trees faster.

4.3.3 Methods for different attributes

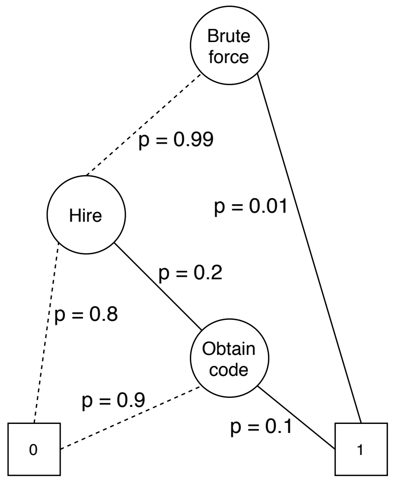
In this section we will specify per attribute how the BDD of an attack tree can be used to analyse that specific attribute.

Probability

Suppose every basic attack step of an attack tree has a certain probability to be (successfully) executed by an attacker. Recall from earlier that the probability for a certain attack A to be successfully executed is the product of the probabilities of all basic attack steps in A. We want to know the total probability that the system which is modelled will be attacked by an attacker. For all possible status vectors of the attack tree we calculate the probability and we sum those together. Since there are shared subtrees, the bottom up method does not work very well as explained earlier in Section 4.3.1. Therefore the BDD method is used. If we want to apply the BDD method, first the attack tree must be translated to a BDD. Every intermediate node in the BDD represents some basic attack step. Along the solid outgoing edge of such a node is the probability that that particular basic attack step is executed successfully (p), whereas along the dashed outgoing edge there is the probability that the attack is not executed successfully (1-p). If we add up the probability of both outgoing edges of a node this should always add up to 1. Figure 4.6 shows an example of an attack tree fragment and the corresponding BDD with probability.



(a) Attack tree tree fragment



(b) BDD of AT fragment with probabilities on edges

Figure 4.6: Example of converting an attack tree to a BDD

The interesting property now to find is the total probability that the system will be attacked successfully. To find the probability of one path, one must multiply all probabilities along the path from the root node to the 1-leaf node. Now to get the probability that any attack is successfully executed we need to add up the probabilities from all paths in the BDD. We do this with an algorithm that uses the topological ordering.
A topological ordering of a graph is a linear ordering of the nodes of the graph such that for every directed edge (u, v) u comes before v in the ordering. In other words, every parent of a node v comes before that node v in the topological ordering. This ordering can be found using several algorithms. One easy way to find it which is also applicable in this case is by using the DFS algorithm, as shown in Figure 4.7.

Call the function FindTopOrder with u as the root node of the BDD. G is the graph, so the BDD in this case, and orderList is an empty list which will contain the topological order at the end. For every node keep track if it has already been visited. This is done by the Boolean variable discovered which every node has. As soon as a node is visited, set the discovered variable to true. Then for every outgoing edge the algorithm makes a recursive call if the new node is not already discovered. The function calls itself, but now with the new node as source node. After this is done for all children of the

```
INPUT
A BDD G with one root node u

OUTPUT
A list containing the reversed topological ordering of the BDD G

METHOD
FindTopOrder(G, u, orderList)
    u.discovered = true
    for every outgoing edge (u, v):
        if(not v.discovered)
            FindTopOrder(G, v, orderList)
        Add u to orderList
        Return orderList
```

Figure 4.7: Pseudo code for finding the reverse topological ordering

node u, so it has done this for all outgoing edges, u will be added to the list and the list is returned. Since this is basically DFS with some small additions, which have no influence on the time complexity, the running time will be the same as DFS. The running time thus is $O(|V| + |E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges.

Correctness: A node is added to the list when all recursive calls are done. This means that all of its children are already added to the list, because the recursive calls of its children are finished. But these children also make recursive calls, which also need to be finished. Therefore a node u is only added to the list if all nodes which are reachable from u are added to the list. So when eventually the list will be reversed to get the topological order, any node comes before any of its descendants in the ordering. Specifically this means that every node will not appear before all of its parents have appeared in the topological ordering and that is exactly what we want.

Example 11. *Recall the BDD we already saw earlier in fig. 4.8. We start at the root node with variable x_1 and call the function FindTopOrder for all its children that are not yet discovered. These are the 0-leaf node and x_2. The 0-leaf node has no outgoing edges, so it is added to orderList and this is returned. The root node then calls the FindTopOrder function with x_2 as node and the orderList consists of the 0-leaf node. x_2 then first calls FindTopOrder for the x_3 node (as we handle the children from left to right in this example). This node has two children of which one, the 0-leaf node, is already discovered hence that node will not be considered. The 1-leaf node is called with the FindTopOrder function, but has no children so it will be added to orderList immediately. Now all calls of x_3 are also finished, so x_3 is added to orderList. x_2 now still has one outgoing edge not yet handled, but this leafs to the 1-leaf node which is already discovered. Therefore, all recursive calls of x_2 are finished and is added to the orderList. Finally, x_1 will be added to orderList and we have the result, which is the following* *list:* $[0, 1, x_3, x_2, x_1]$. *When reversing this list, we indeed have found the topological ordering.*

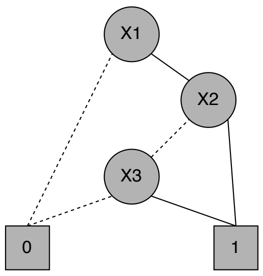
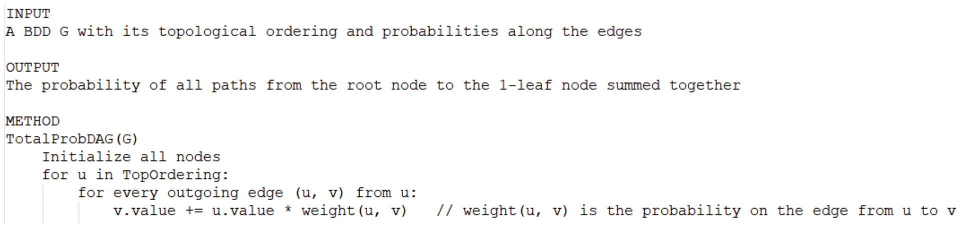


Figure 4.8: A BDD of the function $f(x_1, x_2, x_3) = x_1 \cdot (x_2 + x_3)$

Once the topological ordering is found, we can use the algorithm as described in Figure 4.9.



```
INPUT
A BDD G with its topological ordering and probabilities along the edges

OUTPUT
The probability of all paths from the root node to the 1-leaf node summed together

METHOD
TotalProbDAG(G)
    Initialize all nodes
    for u in TopOrdering:
        for every outgoing edge (u, v) from u:
            v.value += u.value * weight(u, v)    // weight(u, v) is the probability on the edge from u to v
```

Figure 4.9: Algorithm to find the total probability of all paths from the root node to the 1-leaf node

All nodes have one variable called value, which corresponds to the probability that that node will be reached via any path from the root node. So eventually, when we obtain this value for the 1-leaf node, we know the value that the 1-leaf node will be reached via any path which is exactly what we want to find. The algorithm is based on the following principle: when you know the probability that any parent of a node u is reached (the values at the parent nodes) and you know the probabilities of the parent nodes reaching u (the edge values from the parent nodes to u), then you can calculate the probability that u is reached via any of its parent nodes.
When initializing all nodes, the value of the root node is set to 1, and the rest of the values are set to 0. This is because the total probability is 1 and we are going to divide this over all paths there are. Then in topological order, every node propagates its value down over its children. This is done by multiplying its own value with the edge weight (being the probability of taking that route) and adding that value to the value of its child. This is done in topological ordering, so that we know for sure that when a node u is propagating its value over its children, u actually contains the correct value. We can guarantee this, because every parent of u has already propagated

its value to u since they appear before u in the topological ordering. When propagating a value down to a child, it is added to the child's value, because all paths are a disjoint set of status vectors, so their probabilities can be added. When this algorithm is completed, the 1-leaf node will contain the probability that it is reached via any path starting in the root node.

Correctness: The correctness of this algorithm is already shown in [16]. It comes down to the following:

- Every path in the BDD leading to a 1-leaf corresponds to a set of status vectors that form a valid attack
- The probability for that set of status vectors to occur is obtained by multiplying their probabilities
- Every path in the BDD corresponds to a disjoint set of status vectors. Hence, their probabilities can be added.

Run time analysis:
We loop through all nodes with running time $O(|V|)$. Furthermore, we walk along each edge once with running time $O(|E|)$. This gives us a running time of $O(|V|) + |E|)$.

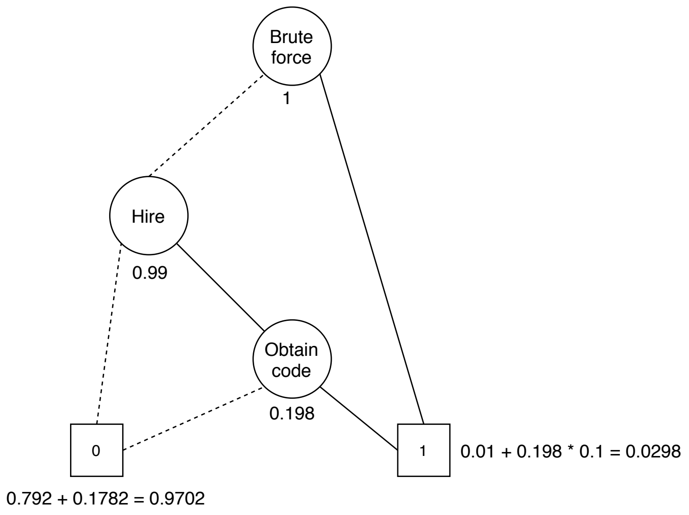


Figure 4.10: Example of the probability algorithm. The value along a node is the probability of getting to that point given the edge probabilities. Topological ordering: [Brute force, Hire, Obtain code, 0, 1]

Example 12. *An example of this algorithm can be found in Figure 4.10. To find the probability that any attack is successfully executed, we need to know the value at the 1-leaf node. We start at the top node with a value of 1. This value will be propagated downwards via both its outgoing edges. The* *edge from 'Brute force' to the 1-leaf node has a value of 0.01, so the value which goes to the 1-leaf node is 0.01. The other edge has a probability of 0.99, so the value that goes to 'Hire' is 0.99 as can be seen beneath it. The next node in the topological ordering is 'Hire'. It has a value of 0.99 and the edge from 'Hire' to 'Obtain code' has a value of 0.2. So the value which goes to 'Obtain code' is* $0.99 \cdot 0.2 = 0.198$*. The value which goes to the 0-leaf node is* $0.99 \cdot 0.8 = 0.792$*. 'Obtain code' is the next node in the topological ordering. It has a value of 0.198 and the value along the edge from 'Obtain code' to the 1-leaf node is 0.1, so the value that will be propagated downwards is* $0.198 \cdot 0.1 = 0.0198$*. The 1-leaf node already contained a value of 0.01, so the new value will be added to get a value of 0.0298. The value which is propagated down to the 0-leaf node is* $0.198 \cdot 0.9 = 0.1782$*. This node already contained a value of 0.792, so the new value will be* $0.792 + 0.1782 = 0.9702$*. Both the 0- and 1-leaf node do not have any outgoing edges, so the algorithm is finished now. As you can see, the value at the 1-leaf node is equal to the value calculated earlier in Figure 4.6a. Also, the value at the 0- and 1-leaf node add up to 1, which is indeed correct because every path ends in either one of those nodes and we started with a probability of 1.*

Cost

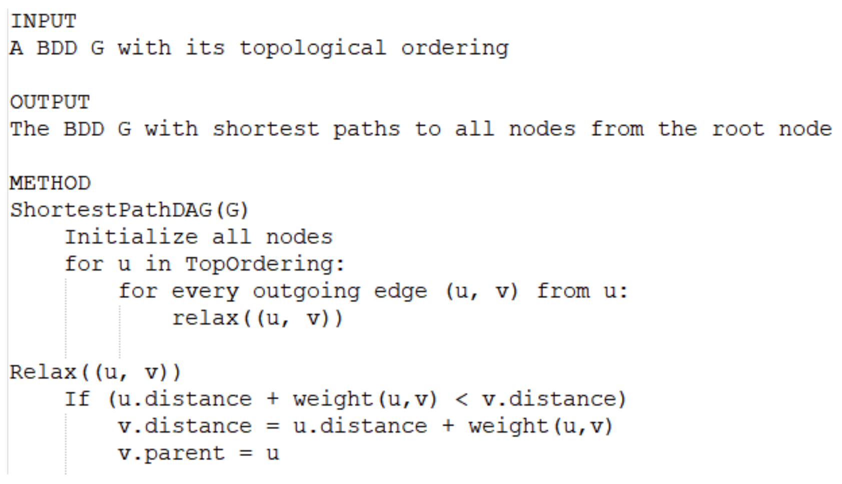
Suppose that every basic attack step has a certain cost to be executed. We assume here that the cost is fixed and is not dependent on time, skill level or any other attribute. Recall that the cost of one attack will be the sum of the costs of all basic attack steps in the attack. When constructing the BDD for this attribute we use the value as follows. Along the solid outgoing edge of a node n is the cost of the basic attack step attached to that node n. Along all dashed edges is a value of 0, because not executing a basic attack step does not cost anything. In contrast to probabilities, it is important that this BDD is a so called MBDD. The BDD only represents minimal cut sets. In practice, an attack is only an attack when the cut set of basic attack steps is also minimal. No attacker would do certain basic attack steps which are not necessary for the attack to be completed. So when analysing the costs of attacks, if there would be an attack in the BDD which is not a minimal cut set, the cost of this attack would make no sense. With probabilities this did not really matter, because every solid edge with a probability p had a counterpart, namely a dashed edge with probability 1-p. Furthermore, we wanted to know the total probability, so all status vectors - minimal or not - had to be involved in the calculation. Here this is not the case, because every dashed edge will standard have a value of 0. So, now we have a MBDD with along the solid edges the cost of the different basic attack steps and along the dashed edges a cost of 0. Again, every path from the root node to the 1-leaf node is an attack. Now the cost of an attack will be the sum of all values along the edges on the path from the root node to the 1-leaf node.

Cheapest attack

One interesting thing to look at is the cheapest attack of an attack tree [15]. In the BDD this means that we want to find the shortest path from the root node to the 1-leaf node. Dijkstra [6] is a well known algorithm which would work perfectly fine for this. However, since BDD's are directed acyclic graphs hence we can find faster algorithms in this case than Dijkstra.

We require the BDD to be reduced for this algorithm to work. Again, the concept of topological ordering can be used for this algorithm. The general idea of the algorithm is as follows. The input is a BDD with a certain root node, the output will be the shortest path with its cost. First we will calculate the topological ordering of the nodes in the graph. Following this the topological ordering will be used to compute the shortest path to every other node in the graph from one starting node with a single source shortest path algorithm for DAGs.

We can use the same algorithm to compute the topological ordering as before, see Figure 4.7. Once the topological ordering is found we can work with the following idea. If we want to know the shortest route from a node S to a node T and we know the shortest paths from S to all parents of T, then the shortest path from S to T is the following. For all parents u of T, take the value of the shortest route from S to u and add the weight of the edge from u to T to it. Then the minimum of all these values is the shortest path from S to T. So if we start this process with the root node of the BDD and do this for every node in topological order then we can calculate the shortest route from the starting node to all other nodes in the BDD. This is also known as the single source shortest path algorithm for DAGs. The implementation of this can be found in Figure 4.11:



```
INPUT
A BDD G with its topological ordering

OUTPUT
The BDD G with shortest paths to all nodes from the root node

METHOD
ShortestPathDAG(G)
    Initialize all nodes
    for u in TopOrdering:
        for every outgoing edge (u, v) from u:
            relax((u, v))

Relax((u, v))
    If (u.distance + weight(u,v) < v.distance)
        v.distance = u.distance + weight(u,v)
        v.parent = u
```

Figure 4.11: Implementation of shortest path in a DAG

The initializing part is pretty straight forward. Set the distance of all nodes

to infinity except for the source node, which should have a distance of 0. Furthermore set the parent of all nodes to nil.
The nodes will be taken in topological ordering starting with the source node. Every outgoing edge of the current node in the algorithm will be relaxed. Of the current node we already know the shortest distance so the shortest path to all its children via the current node can be calculated using the Relax function. The distance of the new path is calculated and compared with the already known distance at v. If the already known distance at v, via another parent of v, is shorter then nothing will happen. However, if the new path is shorter than the currently known shortest path to v, then the value of distance will be updated and the current node u will be set as the parent of v. Once this is done for all nodes all edges are checked and the shortest path from the source node to all other nodes is found.
So the shortest path from the root node to the 1-leaf node represents the cheapest attack of all attacks which are possible according to the BDD. Also because every node keeps track of its parent, we can reconstruct what the actual shortest path is and not only the length of it. This can easily be done by backtracking starting from the 1-leaf node all the way up to the source node. Note that there could potentially be other attacks with the same cost. If one would like to find all these attacks, then a node should keep track of more parents in case there are distances which are equal when comparing in the relax function. Furthermore, if someone for a reason wants to know the most expensive attack, all costs in the BDD can be negated and this algorithm can be executed. The result will be the most negative value, so when negating it back it will be the highest value hence it is the most expensive attack.

Correctness: The algorithm is already an existing algorithm, so we can guarantee that with this algorithm we do find the shortest paths from one node to all other nodes in the graph. It also returns the cheapest attack of an attack tree T, because we constructed a MBDD of T, which only consists of minimal cut sets, thus actual attacks. So every path from the root node to the 1-leaf node in the MBDD is the status vector of a minimal cut set. Given the fact that the MBDD models the behaviour of the structure function, every minimal cut sets of T will also be part of the MBDD. Therefore, we can conclude that this method is correct, because we find the cheapest path out of all minimal cut sets.

Run time analysis:
We need to loop through all the vertices with running time $O(|V|)$. Then we need to check all outgoing edges at all nodes (edge relaxations) with running time O($|E|$). This gives us a running time of $O(|V| + |E|)$. We already saw that finding the topological ordering has the same running time as DFS: $O(|V| + |E|)$. Therefore the running time of the whole algorithm will be $O(|V| + |E|)$, which is pretty efficient and also more efficient than Dijkstra's

algorithm which has a running time of $O((|E| + |V|) \cdot log|V|)$ (when using a binary heap or priority queue).

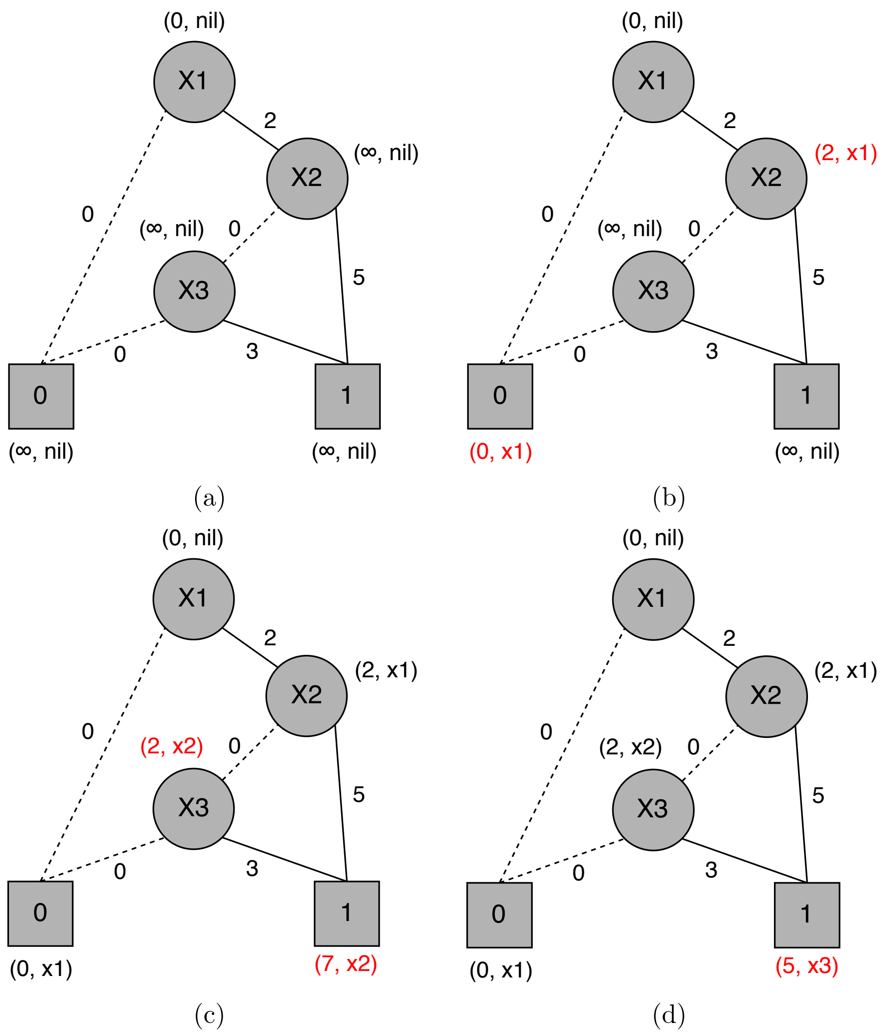


Figure 4.12: Snapshots of the shortest path algorithm for cost. (n, x) next to a node represents (the cost of the shortest path, the parent node). The topological ordering is $[x_1, x_2, x_3, 1, 0]$

Example 13. *Figure 4.12 shows an example of the shortest path algorithm. The values along the edges are the cost of a certain basic attack step and the tuples (value, parent) are part of the algorithm. As you can see, the values along the dashed edges are 0 as we discussed earlier. We can also see that the cost of basic attack step x_1 is 2, x_2 has a cost of 5 and the cost of x_3 is 3. First we need to find the topological ordering, which is $[x_1, x_2, x_3, 1, 0]$ as we saw earlier.*

- *Figure 4.12a is the starting point after initializing. The source node x_1 has a distance of 0 and all other nodes have a distance of ∞. Furthermore, every node has its parent set to nil.*
- *Figure 4.12b shows the situation after relaxing all outgoing edges of the first node in the topological ordering, x_1. Everything that has changed is marked in red.*
- *Continuing this process results in the situation as it is in Figure 4.12d. Here we can see that the shortest path, thus the cheapest attack, has a cost of 5. Backtracking shows that this path is $x_1, x_2, x_3, 1$. In this path x_1 and x_3 are solid lines and x_2 is a dashed line, meaning that the corresponding attack consists of the basic attack steps x_1 and x_3.*

All attacks given a maximum cost

It could also be possible that someone does not only want to know the cheapest attack, but all attacks cheaper than a certain cost. There does not exist a polynomial time algorithm to do this. The number of attacks given a constraint, for example all attacks cheaper than 10.000 euros, can be exponential in size. This means that in the worst case the algorithm also would have an exponential running time. Imagine that someone wants to know all attacks with a cost below a certain threshold, but this threshold is set so high, then this means that even though we can represent all minimal cut sets compact in a MBDD, all paths of the MBDD would satisfy this condition, which means an exponential number of paths given the number of nodes in the MBDD.

To find all attacks cheaper than a certain cost, we can extend the shortest path algorithm in a DAG. The idea of the algorithm stays the same, first finding the topological ordering and then calculating the distances.

INPUT: The input stays the same, a BDD G with one root node u.

OUTPUT: The output will not only be the shortest path, but all path satisfying the condition, thus all paths with a lower cost than the given limit.

METHOD: With the shortest path algorithm we only stored the shortest distance to a node, but now we need to store all distances to a node which are smaller than the given constraint. Every distance also keeps track of its own parent. This parent pointer consists not only of the parent node, but also which value of the parent node is the predecessor. This is needed in order to be able to backtrack and find the path corresponding to the path length. For example if at a node u there are 10 values for paths, and u is a predecessor of a value at v, then v also needs to know which of those 10 values is the parent in order to be able to backtrack and find the path corresponding to that path cost. Another modification will be in calling the relax function. In the shortest path algorithm we would call the relax function once for every outgoing edge at a node, but this was the case because we only stored one distance, namely the shortest distance. Now we store the distances of all paths which are below a certain limit, so for every value at a node we need to relax all outgoing edges at that node. Because the number of values at a node can grow exponentially, it could also be the case that we need to call the relax function a lot. We can try to minimize this by first pruning part of the input BDD. Theoretically this won't change anything in the running time complexity, which will still be exponential, but in practice it can make a difference.

Correctness: The whole idea of the algorithm is exactly the same and works on the same principles. If you know the distances to all parent nodes of a node u and you know the edge weights from those parent nodes to node u, then you can calculate all paths + costs to node u. So although we changed the algorithm slightly, we can guarantee that it still works correctly.

Further optimizations

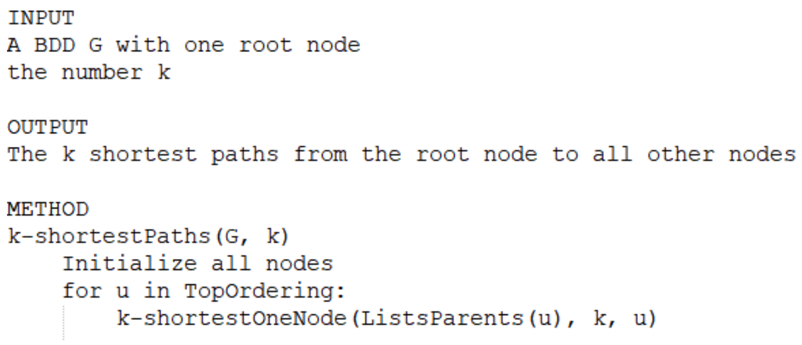
If you want to find all paths with a cost lower than x, then the first thing you can do for pruning is deleting all edges which have a cost of x or more. Any attack via that path will have at least a cost of x which is something you want to avoid since you are looking for all paths with a cost lower than x.

The shortest path algorithm in a DAG can also be used for pruning so that some attack paths can already be discarded. If we perform the shortest path in a DAG algorithm from the 1-leaf node, with all edges in the opposite direction, then we calculate all shortest paths from the 1-leaf node to all other nodes (except the 0-leaf node since that one is not reachable from the 1-leaf node). If at a non terminal node the lowest cost to get to the 1-leaf node is already higher than x, then we know for sure that any attack via this node is not cheaper than x. Therefore this node can be deleted from the BDD along with all edges connected to it.

K shortest paths

If a bound k is selected then it is possible to find the k shortest paths in polynomial time, for example if one wants to know the 10 cheapest attacks. Now we know that the output will not be of exponential size, but only a maximum of 10 attacks. One algorithm which can be used for this is the one described by Eppstein in [8]. However, this describes how to find the k shortest paths in a directed graph whereas we are dealing with directed acyclic graph.

Therefore we use the same concept as earlier, topological ordering. This can be done in the same way as earlier, see Figure 4.7. Figure 4.13 shows the implementation of the k shortest paths in a DAG. Instead of relaxing edges, this algorithm calculates the k shortest paths of all nodes in topological ordering. When the algorithm arrives at a node, the k shortest paths to all its parents are already known and therefore the k shortest paths for the current node can also be calculated.

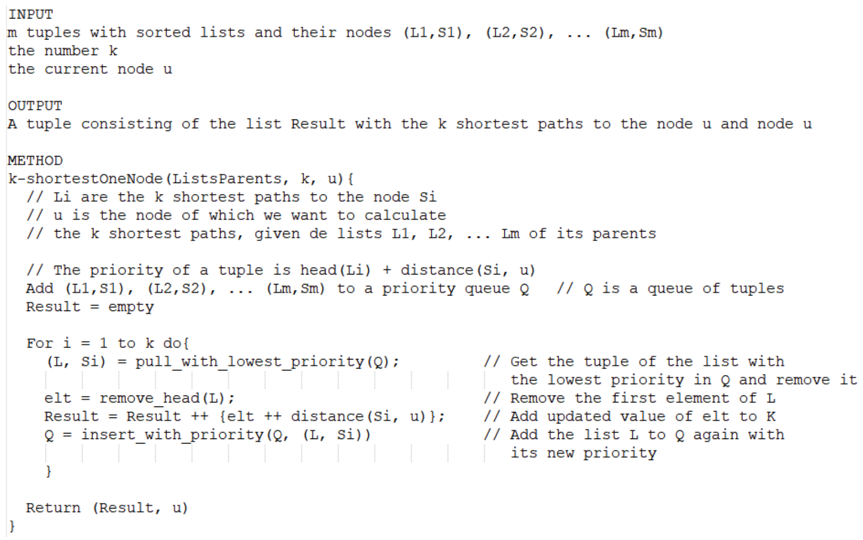


```
INPUT
A BDD G with one root node
the number k

OUTPUT
The k shortest paths from the root node to all other nodes

METHOD
k-shortestPaths(G, k)
    Initialize all nodes
    for u in TopOrdering:
        k-shortestOneNode(ListsParents(u), k, u)
```

Figure 4.13: implementation of k shortest paths in a DAG



```
INPUT
m tuples with sorted lists and their nodes (L1,S1), (L2,S2), ... (Lm,Sm)
the number k
the current node u

OUTPUT
A tuple consisting of the list Result with the k shortest paths to the node u and node u

METHOD
k-shortestOneNode(ListsParents, k, u){
  // Li are the k shortest paths to the node Si
  // u is the node of which we want to calculate
  // the k shortest paths, given de lists L1, L2, ... Lm of its parents

  // The priority of a tuple is head(Li) + distance(Si, u)
  Add (L1,S1), (L2,S2), ... (Lm,Sm) to a priority queue Q   // Q is a queue of tuples
  Result = empty

  For i = 1 to k do{
    (L, Si) = pull_with_lowest_priority(Q);         // Get the tuple of the list with
                                                       the lowest priority in Q and remove it
    elt = remove_head(L);                          // Remove the first element of L
    Result = Result ++ {elt ++ distance(Si, u)};   // Add updated value of elt to K
    Q = insert_with_priority(Q, (L, Si))           // Add the list L to Q again with
                                                       its new priority
    }

  Return (Result, u)
}
```

Figure 4.14: implementation of k shortest paths for one node

Figure 4.14 shows how the k smallest values at one node are calculated if the lists of all its parents are given. We know that all lists from the input tuples are sorted. All input tuples are then put into a priority queue, where the priority is based on the lists of the tuples. The priority takes the edge weight between a parent node and the current node into account, because the priority of a tuple (L_i, S_i) is the head $L_i + weight(S_i, u)$ where u is the current node of which we want to calculate the k shortest paths. Since the lists itself are sorted and the tuples in Q are also ordered according to the priority, we can conclude that the first element of the list of the first tuple in Q is the shortest path there is to node u. Therefore the first list tuple (L, S_i) will be pulled from Q and then the first element *elt* from L is removed and obtained from L. The updated value of *elt* is then added to the output list Result. The tuple (L, S_i), which now has another priority because $head(L)$ is different, will again be added to Q in the correct place according to its priority. This process of obtaining the first element of the list of the first tuple will be repeated k times to obtain the k shortest paths at a node. After these k iterations we can guarantee that we have indeed the k shortest paths, because in each iteration we add the smallest value at that moment to k. Furthermore, each value also keeps track of its parent to be able to backtrack a path.

Correctness: Calculating the k shortest path for one node given the paths to its parents is correctly done in this way, because for k iterations it every time takes the first element from the list with the lowest priority.

- All parent lists are sorted with the lowest cost being in front, $Head(L) = \min L$. These lists are part of a tuple and all tuples are stored in Q. The priority of a tuple (L_i, S_i) in Q is $Head(L) + weight(S_i, u)$ where u is the node of which we want to calculate the k shortest paths. Because every list L_i is sorted, this is the cheapest cost to get to node u via node S_i. Q contains tuples for all parents, so when sorted on the above priority, it is sorted on the cheapest costs to get to node u via any parent node S_i. Therefore, if adding the first element from the list of the first tuple in Q, we obtain the shortest path to node u via any parent node in this iteration.

- The first element is deleted from the list of the first tuple, and the tuple, with an updated list because the first element is removed, is inserted back into the priority queue. Because it is inserted back into a priority queue, it will automatically be placed in the correct position.

- Next iteration, the same process is repeated and for the same arguments we can guarantee to get the shortest path of that iteration.

- This is repeated k times, and every iteration we take the shortest path. From this we can conclude that we obtain the k shortest paths.

The fact that we obtain the k cheapest attacks from this method is based on the same arguments as earlier when we were looking for the cheapest attack. We search for the k shortest paths in the MBDD which only shows minimal cut sets of an attack tree, which guarantees that the k shortest paths will be equivalent to the k cheapest attack of an attack tree (only counting minimal cut sets as an attack).

Run time analysis:
First we will analyse the running time of the computation at one node followed by the running time of the whole algorithm. Suppose that the number of parents of a node i, and thus the number of input tuples, is M_i. The first thing which is done is constructing a priority queue Q with all input tuples. Using Floyd's heap construction algorithm this can be done in $O(M_i)$ time. Creating an empty list Result can be done in constant

time. Now the algorithm reaches the for loop. Pulling the top element of a priority queue requires $O(logM_i)$ time. Removing the first element of a list can be done in constant time. Adding the updated value to the end of Result can also be done in constant time if we keep track of a pointer to the end of the list. Finally inserting the tuple (L, Si) back into Q requires $O(logM_i)$ time again. This is done k times, which gives a running time of $k \cdot 2logM_i = O(k \cdot logM_i)$.
So constructing Q has a running time of $O(M_i)$. The number of parents is also M_i, meaning that the number of incoming edges is M_i. Thus the construction algorithm needs $O(1)$ time for every edge, meaning that for the overall algorithm the running time for this part will be $O(m)$ where m is the number of edges in the BDD.
Furthermore, the running time of the for loop part is $O(k \cdot logM_i)$. This has to be done for all nodes in the BDD. Assuming the number of nodes in the BDD is n, we obtain $O(k \cdot logM_1 + k \cdot logM_2 + \ldots + k \cdot logM_n) = O(k \cdot log(M_1 \cdot M_2 \cdot \ldots M_n))$. The maximum is reached when $M_i = m/n$, with m the amount of edges and n the amount of nodes. Now we obtain $O(k \cdot log(M_1 \cdot M_2 \cdot \ldots M_n)) = O(k \cdot log(\frac{n \cdot m}{n})) = O(k \cdot log\ m)$.
Now we can conclude that the total running time is $O(m + k \cdot log\ m)$.

Example 14. *Figure 4.15 shows a BDD fragment on which we perform the k shortest path algorithm at a node. Node S is the node of which we want to know the k shortest paths, where k is 4 in this situation. P_1, P_2 and P_3 are the three parent nodes of S, with their k shortest paths showed in a list above these nodes. The edge weights from P_1, P_2 and P_3 are 1, 2 and 0 respectively.*

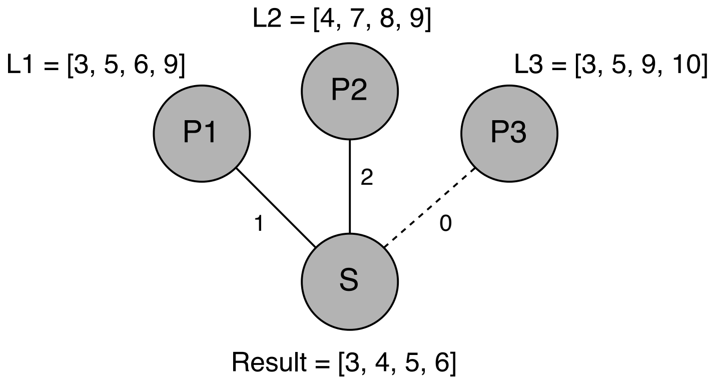


Figure 4.15: Fraction of a BDD fragment showing the situation of this example

The process of the algorithm is shown in Figure 4.16. Initially the list result is empty. Since k is 4, there will be 4 iterations as is displayed in the figure by a-d. In each iteration the top part shows the parent lists in the current

Input lists	L1(4) = [3, 5, 6, 9]	L1(4) = [3, 5, 6, 9]	L1(6) = [5, 6, 9]	L1(6) = [5, 6, 9]
	L2(6) = [4, 7, 8, 9]	L2(6) = [4, 7, 8, 9]	L2(6) = [4, 7, 8, 9]	L2(6) = [4, 7, 8, 9]
	L3(3) = [3, 5, 9, 10]	L3(5) = [5, 9, 10]	L3(5) = [5, 9, 10]	L3(9) = [9, 10]
Priority queue	Q = [L3, L1, L2]	Q = [L1, L3, L2]	Q = [L3, L2, L1]	Q = [L2, L1, L3]
Selected list	L = L3	L = L1	L = L3	L = L2
Selected element	elt = 3	elt = 3	elt = 5	elt = 4
Updated result	Result = [3]	Result = [3, 4]	Result = [3, 4, 5]	Result = [3, 4, 5, 6]
	(a)	(b)	(c)	(d)

Figure 4.16: k-shortest algorithm at node S of Figure 4.15. Since k = 4, there are also four iterations, shown as a-d. On top of every iteration we everytime see the lists as they are during that iteration. The number x in L_i shows the priority of list i. Q shows the priority queue at that iteration (we only show lists here instead of tuples as described in the algorithm for the sake of explaining this example). L is the pulled list from Q and elt is the first element of L. Result is the list at the end of the iteration

situation, with the value of its priority i.e. head(L) + edge weight, between brackets. So the list with the lowest value has the biggest priority. This is reflected in the ordering of the list in Q (for the sake of explaining this algorithm we only show lists in Q. Normally Q would be a list of tuples with a list and a node). Next, the first list from Q will be pulled and is called L. Then the first element from this list L is pulled and displayed as elt. Elt will then be updated with the edge weight and added to Result. After four iterations the four smallest values have been found and the algorithm is finished. Now the 4 smallest paths to node S are also known.

Time

For analysing time, we can again make the division between non-parallel and parallel AND gates.

Non-parallel AND-gate

Suppose every basic attack step has a fixed amount of time which is needed in order for the basic attack step to be executed successfully. Then every solid outgoing edge contains the time to perform that specific BAS and every dashed outgoing edge contains a value of 0 (because not executing a basic attack step also takes no time). It is important that the BDD representing the attack tree only contains minimal cut sets for the same reason as with cost hence the BDD should be a MBDD. The attack steps cannot be done in parallel in this case hence the time it takes for an attack to complete will be the sum of the time values of all basic attack steps in the attack just like with costs. So, in a BDD the time to perform an attack is the sum of all values along the path to the 1-leaf node. To find the shortest attack, all attacks shorter than a certain amount of time or the k-shortest attacks the same algorithms can be used as described earlier for the costs of an attack.

Parallel AND-gate

The structure for the BDD in this case is the same as in the non-parallel case. However, in this case all attack steps can be done at the same time. This means that the attack can be executed way faster. The time for an attack will now be the maximum time value of all basic attack steps in the attack. For example suppose an attack can be done by performing 4 basic attack steps which take 3, 5, 4 and 2 hours to perform then the attack will take 5 hours to be executed. So in a BDD the time of an attack can be found as the maximum of the values along the path to the 1-leaf node.

Shortest attack

When one wants to know the shortest attack in this situation, he needs to find the path in the BDD of which the maximum value along the path is the lowest. In other words, we want to find the most expensive edge on the shortest path since that edge determines the cost of the attack. A concept

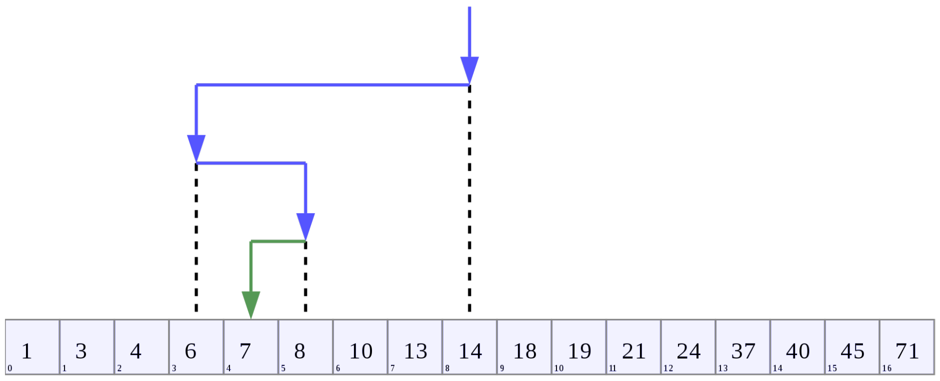


Figure 4.17: Example of binary search looking for the number 7

we will use in the algorithm is binary search. It works on a sorted list to find a certain element. First it checks if the given element matches the middle element of the list. If so, then you found the element you were looking for. If not, then you know that the element you are looking for is either in the left half of the list or the right half of the list since it is an ordered list. You apply the same steps on either the left or right half of the list until you find the element you are looking for. An example of this can be found in Figure 4.17.

The idea of the algorithm is the following. we are looking for the most expensive edge of the shortest path, which we will call e. First all edges will be sorted in descending order according to their weight. Let C be the set of candidates, of which one is the edge e we are looking for. At the start C will contain all edges in the BDD. Then the $\frac{|C|}{2}$ most expensive edges in C will be deleted from the BDD. This can be done quickly, since we sorted the edges in the first step. Now we check if there is still a path from the root node to the 1-leaf node. There are two options:

- If there is still a path, then we know for sure that the set of just deleted edges does not contain the edge e we are looking for, because there is still a path with only edges that are not more expensive. So these edges can also be deleted from the set C.
- If there is not a path anymore, then we can conclude that the edge e is one of the just deleted edges. We need to add the just deleted edges back to the BDD, and the set C will now only contain the edges which were just deleted. Note that the cheapest edges still need to be part of the BDD, otherwise checking for a path in the BDD might not work.

This previous step will be repeated until there is just one edge left in C, which will be the edge e we are looking for. Now all paths that are left will have to pass through this edge e and thus are all attacks that will cost the same amount of time. These are also all the shortest attacks time wise.
In summary, the algorithm works as follows:
INPUT: A BDD G with one root node
OUTPUT: The critical edge that determines the cost of the shortest attack plus a partial BDD showing the shortest path.
METHOD:

1. Sort the edges in descending order according to their weight.
2. Add all edges to the set C
3. Delete the $\frac{|C|}{2}$ most expensive edges in C from the BDD
4. Check for a path in the BDD.
   If there is one, delete the just deleted edges also from the set C. If C now only contains one edge, then edge e is found. Otherwise, go back to the previous step.
   If there is no path, add the just deleted edges back to the BDD and C now only consists of the just deleted edges. If C now only contains one edge, then edge e is found. Otherwise, go back to the previous step.

Correctness: Essentially, we are looking for one edge, namely the most expensive one on the shortest attack since that is the edge which determines the cost of the shortest attack. In step 1, all edges are sorted in descending order, so from the most expensive edges to the cheapest edges. We keep track of a set of candidates of which one will be the edge e we are looking for. Every iteration the most expensive edges are deleted, so when the point is reached where there is only one edge left, we can conclude that we did not delete a cheaper edge which could have lead to a faster attack. Furthermore, every iteration there is a check for a path hence every time we know in what half of C the edge e is located. With these two observations, it has to be the case that when there is just one edge left in C this is the one we are looking for. Thus, we have found the most expensive edge on the shortest path, which tells us the cost of the fastest attack.

Checking for a path in the BDD
Checking for a path from the root node to the 1-leaf node can either be done by using the BFS or the DFS algorithm. Since we want to know if there is a path from the root node to the 1-leaf node and the 1-leaf node is deep in the graph it is better to use the DFS algorithm. We start DFS on the BDD from the root node. We let DFS run and whenever the algorithm encounters the 1-leaf node it should return true. If the DFS algorithm is finished and it did not encounter the 1-leaf node then there is no path from the root node to the 1-leaf node and it should return false.

Run time analysis:
For sorting several algorithms can be used. Depending on the preferences a sorting algorithm can be chosen. Looking at run time complexity, merge sort, heapsort and quicksort are all a good option, with running time $O(|E| \cdot log|E|)$. Deleting the $\frac{|C|}{2}$ most expensive edges in C can be done in linear time, being $O(|E|)$. Checking for a path using DFS can be done in $O(|V| + |E|)$ time. This means every iteration takes $O(|V| + 2|E|) = O(|V| + |E|)$ time. Since we use the idea of binary search, there will be a maximum of $Log|E|$ iterations, giving a running time of $O((|V| + |E|) \cdot Log|E|)$. The running time of sorting does not matter compared to this, so the running time of the algorithm will be $O((|V| + |E|) \cdot Log|E|)$.

Example 15. *Figure 4.18 shows snapshots of how this algorithm works. In Figure 4.18a the starting point is shown. On the right side of each BDD the set of edges, in descending order, is shown. The curly bracket around it marks which edges are in the set of candidates C. We start with deleting the $\frac{|C|}{2}$ most expensive edges in C, which is shown in Figure 4.18b. There is no path from the root node to the 1-leaf node anymore hence the just deleted edges will be added back to the BDD and the set C now consists of the just deleted edges, shown in Figure 4.18c. Again, the $\frac{|C|}{2}$ most expensive edges in C will be deleted (because half of the edges would be 2.5, one can choose to delete either 2 or 3 edges, which is 2 in this example). This results in the BDD shown in Figure 4.18d. The red marked edges at the right side are not part of the BDD anymore. There is still a path from the root node to the 1-leaf node, so we can leave the just deleted edges out and remove them from C. Again, edges will be deleted (just one edge now) resulting in Figure 4.18e. There is not path left anymore, so we add the edge back to the BDD and we update C. In Figure 4.18f there is only one edge left in C, meaning that the critical edge is found. This edge is marked in green and by eyeballing this small example one can indeed verify that this is correct. The shortest path is A, C, E, 1 with a cost of 3 because of the edge (E, 1).*

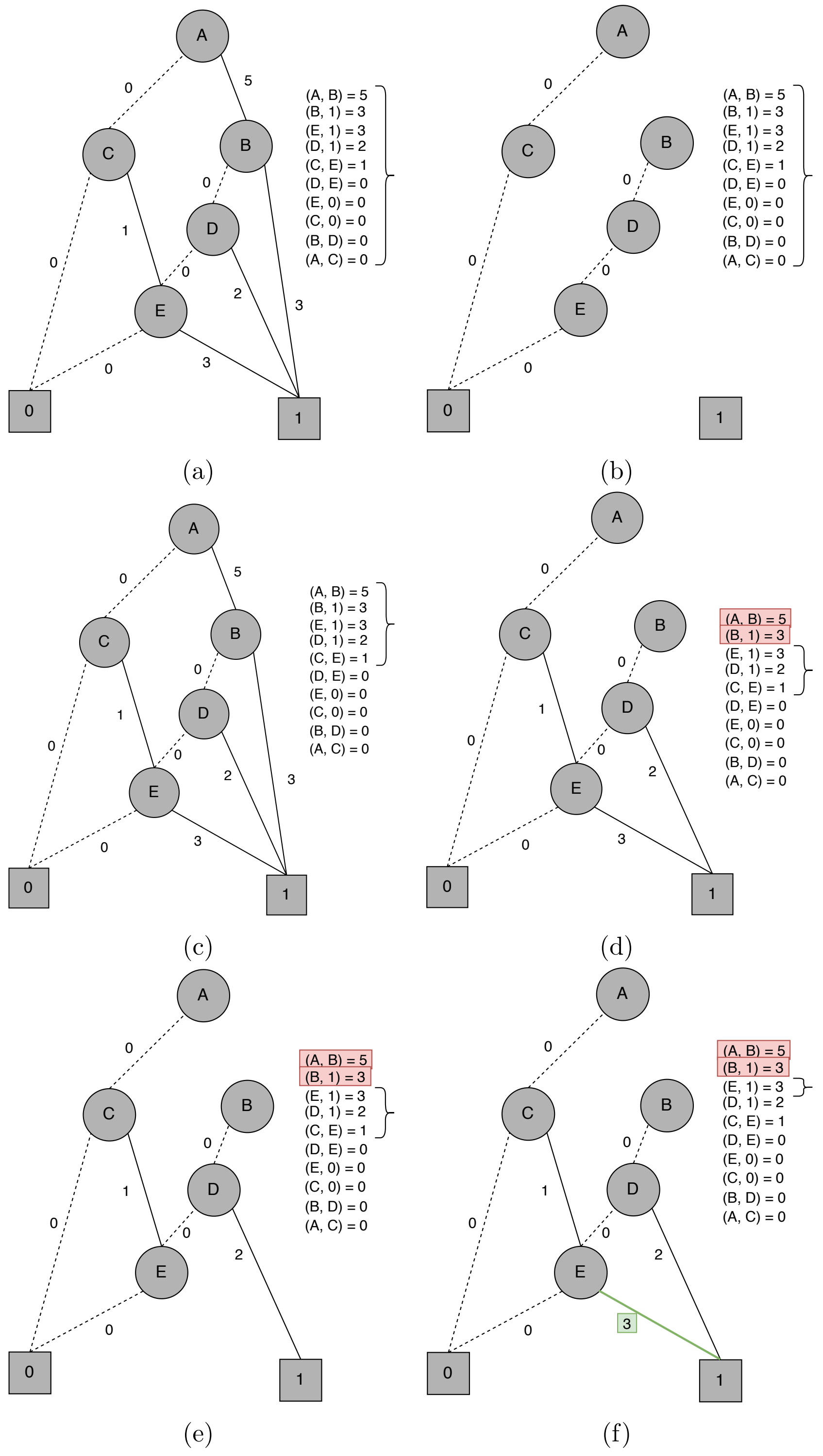


Figure 4.18: Snapshots of the shortest path algorithm for time. Along each figure all edges can be found. All edges in the range of the curly bracket are part of the set C. The red marked edges are permanently removed from the BDD at that stage

All attacks faster than a given time
If one wants to find all attacks which are faster than a given time x, the following algorithm can be used.
INPUT: A BDD with one root node
OUTPUT: A partial BDD with only paths faster than the given time
METHOD: We start with the BDD corresponding to the attack tree with the times of the basic attack steps along the edges. Now loop through all the edges and if an edge has a value which is greater or equal to x, delete this edge. In this way the BDD will only contain edges with a lower value than x, meaning that every path in the BDD takes at most $x - 1$ time. Note that after deleting some edges, the result is not a BDD anymore but rather a partial BDD. It could also be the case that some nodes are not reachable anymore.
Correctness: All edges with a cost that is too high will be deleted. Because the time an attack takes will only be the maximum of all edges on the path from the root node to the 1-leaf node and there is no edge which is more expensive than the given threshold, all paths will not exceed this threshold.
Better alternative METHOD: It would be more useful if the output would be a proper BDD, also when using this algorithm in combination with other leaf attributes. This means that the BDD should also be reduced afterwards. This whole process then ends up to be the same as the Restrict function described by Bryant in [4]. Arguments for this function are variables of the BDD which all also get a truth value. What this function then does is traversing the BDD and when it reaches a variable that has been given as argument, it sets that variable to the given value hence it restricts some variables to a certain value. In our case we would give every basic attack step with a time of x or higher as an argument and set all their values to 0. In this way all basic attack steps with a cost of x or higher are restricted to the value false, meaning there will only be path with value along the edges which are smaller than x.
Run time analysis:
The first variant, the only thing that happens is looping through the edges and removing only those edges with a cost of x or higher. Removing an edge can be done in constant time if implemented with the correct data structure. Therefore the running time would be $O(|E|)$.
The Restrict function by Bryant has a running time of $O(|G| \cdot log|G|)$, where $|G|$ is the size of the input BDD in terms of number of nodes.

Skill level

Suppose there a some skill levels which can be ordered, for example low, medium and high. Every basic attack step has a certain minimum skill level needed in order to be successfully executed. So if a basic attack step has a minimum skill level of medium, then only attackers with skill level medium or high can execute this basic attack step successfully. Recall that the required skill level of an attack is the maximum skil level of all basic attack steps in the attack. The values along the edges in the BDD will be as follows. Every solid outgoing edge contains the minimum skill level needed to execute the basic attack step of the node, and every dashed outgoing edge contains the lowest skill level which exists. The BDD should be a MBDD, because when not working with a minimal cut set, the element that can be left out can have an influence on the minimum skill level needed for that cut set. The minimum skill level needed for a certain attack is the maximum skill level along the path from the root node to the 1-leaf node. So if a path contains the skill levels $\{low, medium, low, low\}$ then every attacker with a skill level of at least medium can execute this attack.

All attacks below a certain skill level

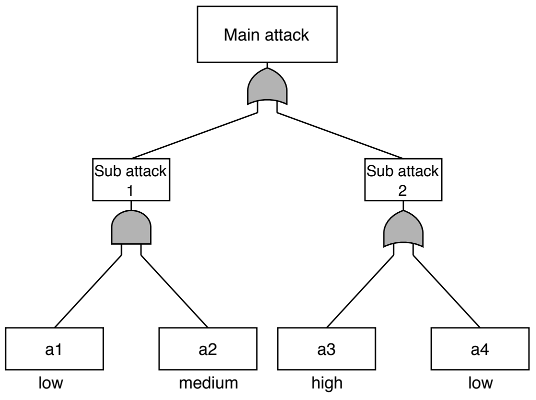
Suppose one wants to find all attacks with a skill level below high. For this task, one can once again use the Restrict function from Bryant [4]. Every basic attack step with a skill level of 'high' or higher should be set to false. So what we give as arguments is all basic attack steps with a skill level of 'high' or higher and the value false. What will be returned is a reduced BDD which only contains path with a maximum skill level of medium in $O(|G| \cdot log|G|)$ time.

All attacks above a certain skill level

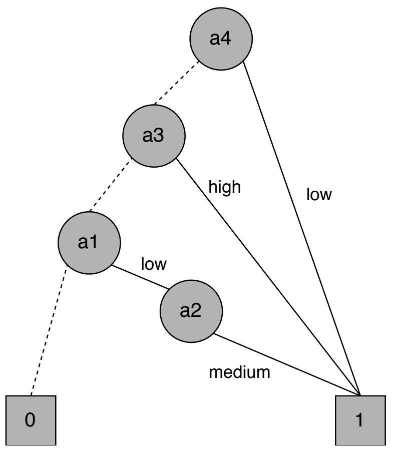
Instead of finding all attack below a certain skill level, suppose someone wants to find all attack above a certain skill level, e.g. all attacks with a minimum skill level of medium. (Until now we did not succeed in finding a suitable algorithm for this yet. However, we tried some things which did not work. We show it here because failure is also a result and others can learn from what we tried)

While Bryants Apply function [4] seems very useful for this, it does not work in this situation. The Apply function takes two BDDs and creates of BDD of them given an operator, for example AND. It seems useful to combine the BDD representing the attack tree with a BDD representing a disjunction of all basic attack steps with skill level of medium or higher using the AND operator. In this way all paths in the newly formed BDD have to satisfy both BDD's, meaning it is a path in the first BDD and also contains at least one basic attack step with skill level medium or higher. In the following example we will show why this does not work.

Example 16. *Consider the attack tree in Figure 4.19a and the corresponding BDD in Figure 4.19b both including skill levels of the basic attack steps. Suppose one wants to know all attacks which require a skill level of medium or higher. This means that every path should contain at least one edge with a skill level of at least medium. When using the Apply method, we combined the BDD in Figure 4.19b with the BDD representing the disjunction of all basic attack steps with a skill level above low, shown in Figure 4.20a.*

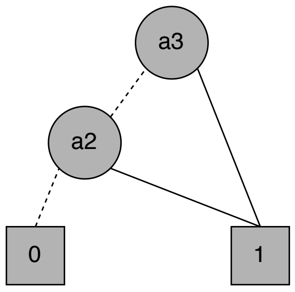


(a) Attack tree structure including skill levels for basic attack steps

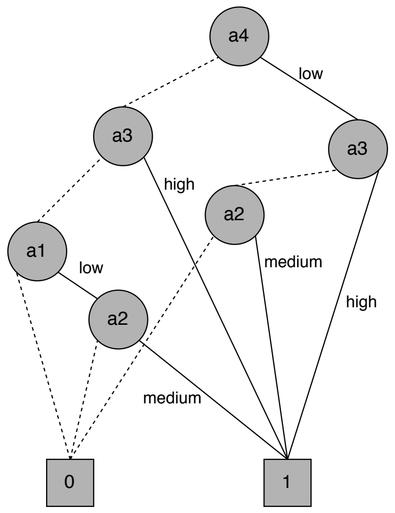


(b) BDD representing the attack tree in Figure 4.19a

Figure 4.19: Attack tree with skill levels and its corresponding MBDD



(a) Disjunction of all basic attack steps with skill level of medium or higher from the AT in Figure 4.19a



(b) BDD after combining the BDD's from Figure 4.19b and Figure 4.20a with the AND operator

Figure 4.20: Helper BDD in Figure 4.20a and resulting BDD after apply function in Figure 4.20b

Combining the BDD's from Figure 4.19b and Figure 4.20a with the AND operator using Bryant's Apply function results in the BDD shown in Figure 4.20b.
In the newly formed BDD, every path does indeed satisfy both the BDD's,

meaning that every path is an attack which also contain a basic attack step with a skill level of medium or higher. However, the problem is that the new BDD is not a minimal BDD anymore. It also shows non minimal cut sets, for example the path (a4, a3, 1) since in Figure 4.19b (a4, 1) was already a valid path. Therefore this newly formed BDD does not have any value, because only minimal cut sets will in practice be real attacks.

Also deleting paths that are not valid in the original BDD does not work. For example, consider the BDD in Figure 4.21. The path (a1, a3, a4, a5 1), in red, is invalid, because that requires a skill level of low. However, (a1, a3, a4, 1), in green, is a valid attack and (a1, a2, a4, a5, 1), in blue, is also a valid attack. To keep these paths in the BDD, no edge on the green and blue path can be deleted. But that means that the invalid path will also remain in the BDD since no edge in the red path can be deleted.

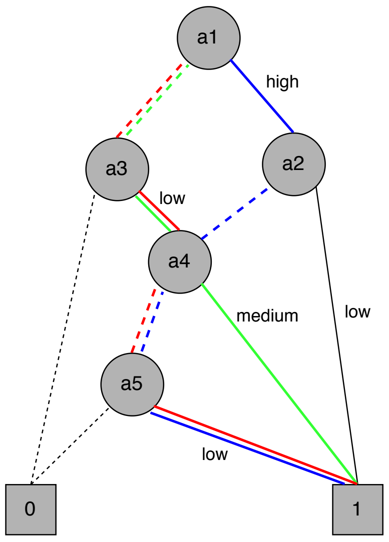


Figure 4.21: A BDD with skill levels showing various paths

Boolean value

This part describes how to analyse Boolean values, which could be any Boolean variable. We will work with special equipment as a Boolean variable, which indicates if special equipment is needed to perform the basic attack step. Every solid outgoing edge contains a true if special equipment is needed to execute the basic attack step and false otherwise. Every dashed outgoing edge contains false since not performing a basic attack step also does not require any special equipment. We require the BDD to be an MBDD for the same reason as with skill. Now if along the path of an attack there is an edge with true special equipment is needed. If this is not the case then no special equipment is needed.

Boolean value can be treated the same way as skill levels, where there would only be two skill levels. Boolean variables like special equipment can also be ordered like skill levels, with false being lower than true. For example, if one wants to find all attacks that do not require special equipment, the Restrict algorithm can be used to restrict the value of all basic attack steps with do require special equipment to false. This is exactly the same as the situation where someone wants to find all attacks with a skill level below high when the set of skill levels exists of $\{low, high\}$.

Combinations

When combining several attributes, you often want to minimize or maximize one attribute and put constraints on the other. For example, you want the cheapest attack, thus minimizing the cost, of all attacks with a skill level below medium, putting a constraint on the set of possible attacks. What we want to do is first creating a BDD with all constraints on it such that afterwards the applicable algorithm can be run to minimize or maximize a certain attribute.
For example, finding the cheapest attack of all attacks with a skill level below medium will go as follows. First the AT is translated to a BDD, which will be followed by the algorithm to find all attacks with a skill below medium as described earlier. This algorithm returns a new BDD G with all attacks that require a skill level below medium. This new BDD G can now be used to run the cheapest attack algorithm on. The result from this algorithm will be the cheapest attack of all attacks with a skill level below medium.
This is not as straight forward when you want to find the fastest attack from the top-10 cheapest attack, since the algorithm for finding the k cheapest attacks does not return a BDD. However, after running the algorithm it is possible to backtrack all k paths and thus knowing the status vectors, i.e. the BAS assignments, of all k attacks. Every path i can be represented as a Boolean function f_i, namely as a conjunction of all basic attack steps that are true for that certain path i. The Boolean function for any of these paths to be true is a disjunction of all paths, so $\bigvee\limits_{i=1}^{k} f_i$. This Boolean function can be represented by a BDD, which can then be used by the algorithms as discussed throughout this section, in this example finding the fastest attack.

Chapter 5

Conclusions

We discussed the analysis of attack trees in this thesis and specifically two types of attack trees. These were attack trees without shared subtrees but possibly including SAND gates and attack trees with shared subtrees but without SAND gates.

Attack trees without shared subtrees can be analysed via the bottom up method, where leaf values from the tree are propagated upwards to the tree to analyse a property of the attack tree. We showed how this can be done for several attributes and properties as different calculation rules are in place for different attributes and properties. It was already a well known and efficient method, so for this we more or less gave an overview of how several attributes can be calculated with this method. However, we extended the bottom up method by also including the SAND gate in it.
For attack trees with shared subtrees, but no SAND gates, we focused on the BDD method. The idea of translating an attack tree to a BDD was already known, but there was barely any analysis on the BDD. It was merely focussed on status vectors and simple risk computations. We showed that this could be extended to analyse different attributes, like cost or skill level. For these attributes, we used/combined already known algorithms to find properties such as the cheapest attack in the attack tree, or all attacks below a certain skill level. Furthermore, we created a new algorithm to find the k shortest paths in the BDD, which corresponds to the k cheapest or fastest attacks for the attributes cost and time respectively. Moreover, we showed how we could combine the analysis of different attributes to be able to make more elaborate queries. For one metric we wanted to calculate, all attacks above a certain skill level, we were not able to find an efficient and working algorithm. We showed what we tried and why this did not succeed.

Future work
The analysis of attack trees with both shared subtrees and SAND gates is left for future work. The bottom up method is not applicable, because those attack trees contain shared subtrees. The BDD method as described in this

thesis is (at least) not directly applicable to attack trees with both shared subtrees and SAND gates. BDDs represent boolean function, but there is no way to show a certain ordering with a BDD, for example sub attack 1 comes before sub attack 2. If this BDD method can be extended by adding more variables to BDD nodes such that it works for attack trees including SAND gates has to be researched in future work. However, we suspect that the BDD method can be used to calculate for example the cheapest cost of the attack tree (not the corresponding attack itself) as SAND gates are handled the same as AND gates when it comes to cost. How this exactly works and for what attributes this can work is also left for future work.
Furthermore, in the analysis of attack trees with shared subtrees and no SAND gates, we assumed that all attributes were fixed. Maybe in real life situations, these values can change once a certain step is completed, for example if one basic attack step is executed, the probability of another one can change if they are not totally independent. The methods we described do not take this dynamic behaviour into account, so that is something left for future work.

Chapter 6

Acknowledgements

I would like to thank my supervisor Mariëlle Stoelinga for helping me constructing this thesis. I could always reach out for help whenever I had questions or wanted feedback. The communication was very good and I appreciate all the help I got!

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[2] Andrea Bobbio. A methodology for qualitative/quantitative analysis of weighted attack trees. pages 133–138, 09 2013.

[3] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In Richard C. Smith, editor, *DAC*, pages 40–45. IEEE Computer Society Press, 1990.

[4] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[5] Ahto Buldas, Peeter Laud, Jaan Priisalu, Märt Saarepera, and Jan Willemson. Rational choice of security measures via multi-parameter attack trees. In Javier Lopez, editor, *Critical Information Infrastructures Security*, pages 235–248, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Jing chao Chen. Dijkstra's shortest path algorithm, 2003.

[7] G.C. Dalton, Robert Mills, John Colombi, and R.A. Raines. Analyzing attack trees using generalized stochastic Petri nets. pages 116 – 123, 07 2006.

[8] David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, February 1999.

[9] 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 Jennifer Horkoff, Manfred A. Jeusfeld, and Anne Persson, editors, *The Practice of Enterprise Modeling - 9th IFIP WG 8.1. Working Conference, PoEM 2016, Skövde, Sweden, November 8-10, 2016, Proceedings*, volume 267 of *Lecture Notes in Business Information Processing*, pages 326–334. Springer, 2016.

[10] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Attack-defense trees. *Journal of Logic and Computation*, 24, 02 2014.

[11] 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, 03 2013.

[12] Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. Quantitative attack tree analysis via priced timed automata. In Sriram Sankaranarayanan and Enrico Vicario, editors, *FORMATS*, volume 9268 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2015.

[13] Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In Dongho Won and Seungjoo Kim, editors, *ICISC*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer, 2005.

[14] L. Piètre-Cambacédès and M. Bouissou. Beyond attack trees: Dynamic security modeling with boolean logic driven markov processes (bdmp). In *2010 European Dependable Computing Conference*, pages 199–208, 2010.

[15] Bruce Schneier. Attack trees - modeling security threats. 1999.

[16] M. Stoelinga and E. Ruijters. Fault tree analysis -an introduction-.

[17] F. Vaandrager. Symbolic model checking with binary decision diagrams, March 2017.