MSc Computer Science

Attack-Defense Trees with Offensive and Defensive Attributes

Danut-Valentin Copae

Graduation committee:
Lopuhaä-Zwakenberg, Milan, dr.
Stoelinga, Mariëlle, prof.dr.
Hahn, Florian W., dr.ing.

July, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

Attack-defense trees (ADTs) are a commonly used methodology for representing the interplay between the attacks on a system and the counter-acting defenses employed to prevent these attacks. ADTs serve as a powerful tool for quantitative analysis, offering a structured approach for prioritizing potential threats and defenses through the use of attributes. Previous work in this domain has only focused on analyzing metrics such as cost, damage, or time from the attacker's perspective. This approach, however, presents an incomplete picture of the system, as it fails to model attributes for the defender: in real scenarios, the defender usually has finite resources for counter-attacks and, just like the attacker, is bounded by some constraints.

This thesis aims to bridge the gap by developing fast algorithms for computing the Pareto front between defense and attack attribute values. Building from the existing mathematical foundations of attack trees from Lopuhaä-Zwakenberg *et al.* (2023), the work extends these models with the concept of attribute domains for the defender. We analyzed tree-structured ADTs using a bottom-up approach and Directed Acyclic Graph (DAG)-structured ADTs (trees with shared sub-trees) using enumerative, Biobjective Integer Linear Programming (BILP) and Binary Decision Diagram (BDD)-based techniques. For a clearer mathematical model and algorithms, we primarily focus on the minimum cost attribute domain for both the defender and the attacker.

The experimental results on random ADTs indicate that rather than finding one algorithm to rule them all, each technique is useful based on varying ADT properties. A bottom-up approach computes the Pareto front the fastest for tree-structured ADTs, while BDDs are the most efficient for DAG-structured ADTs. The implications of our work enable a more detailed analysis of attack scenarios, allowing the system owners to make better-informed decisions.

Keywords: attack trees, attack-defense trees, Pareto front, multi-criteria optimization

Contents

1 Introduction **1**
- 1.1 Attack-Defense Trees . . . 2
- 1.2 Quantitative Analysis on Attack Trees . . . 3
- 1.3 Quantitative Analysis on Attack-Defense Trees . . . 3
- 1.4 Research Goal . . . 4
- 1.5 Cost-Cost Pareto Front . . . 5
- 1.6 Contributions . . . 6

2 Background & related work **7**
- 2.1 Attack Trees . . . 7
  - 2.1.1 Attack-Defense Trees . . . 7
  - 2.1.2 Directed Acyclic Graphs . . . 8
- 2.2 Quantitative Analysis . . . 9
  - 2.2.1 Semirings . . . 9
  - 2.2.2 Quantitative Analysis on Tree-Structured ATs . . . 9
  - 2.2.3 Quantitative Analysis on DAG-structured ATs . . . 10
  - 2.2.4 Quantitative Analysis on Attack-Defense Trees . . . 12
- 2.3 Pareto Front . . . 12

3 Attack-Defense Trees **13**
- 3.1 Syntax . . . 13
- 3.2 Attributes . . . 14
- 3.3 Semantics . . . 16
- 3.4 Pareto Front . . . 17
- 3.5 Minimum Cost Pair . . . 18

4 Tree-structured ADTs **19**
- 4.1 Bottom-up Algorithm . . . 19
- 4.2 Complexity . . . 22
- 4.3 Proof for the Bottom-up Algorithm . . . 22

5 DAG-structured ADTs **26**
- 5.1 Naive approach . . . 27

6 Biobjective Integer Linear Programming **28**
- 6.1 Variables . . . 29
- 6.2 Constraints . . . 29
- 6.3 Objectives . . . 30
- 6.4 Model . . . 30

7 Binary Decision Diagrams **32**
7.1 Evaluating Path Costs . . . 33
7.2 Variable ordering . . . 34
7.3 Algorithms . . . 35
7.3.1 BDD-PATHS . . . 35
7.3.2 BDD-ALL-DEF . . . 36
7.3.3 BDD-BU . . . 37
7.4 Complexity . . . 38

8 Experiments **39**
8.1 Random ADTs . . . 39
8.2 Cloned ADTs . . . 42

9 Conclusion & Discussion **44**
9.1 Future work . . . 44

Appendices **46**

A Experiments appendix **47**

B Random ADT algorithm **48**

C Random Trees Results **49**

Chapter 1

Introduction

Thanks to computerized systems, the healthcare, financial, and business sectors have seen significant technical advancements. However, these systems can become notoriously complex when multiple actors, IT systems, and physical systems are involved. The resulting complexity also raises the number of possible breaches that attackers can exploit. This is especially true with cyber-physical systems, which have delicate interplays between components. Consequently, there is a need for robust and systematic threat modelling systems that can cope with such attacks.

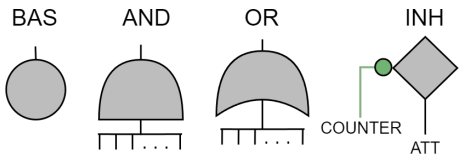


FIGURE 1.1: Legend of the gate types used in the attack-defense tree illustrations.

Fault trees, introduced in the 1960s [1] to evaluate the launch control system of ballistic missiles, were one of the first models used to analyze the safety of a system. Through their hierarchical structure, they model the failure of a system through the failure of individual components. Building on this idea, Schneier [2] introduced attack trees (ATs), which nowadays represent one of the most prominent tools to evaluate the security of complex systems. While safety remains a critical concern, ATs have complemented the traditional focus on accidental internal failures with protection against deliberate, malicious attacks.

The primary utility of ATs is to describe the various strategies an attacker can take to compromise a system through a structured decomposition of the attack into smaller objectives. This enables security experts to design countermeasures for preventing future attacks [3, 4]. Due to their simplicity and compact form, ATs are commonly used in commercial software tools such as Amenaza's SecurITree [5] and Isograph's AttackTree [6] as well as industrial applications, e.g. analyzing the security of a SCADA system for a tank and pump facility [7], impact analysis of electric grid feature scenarios [8].

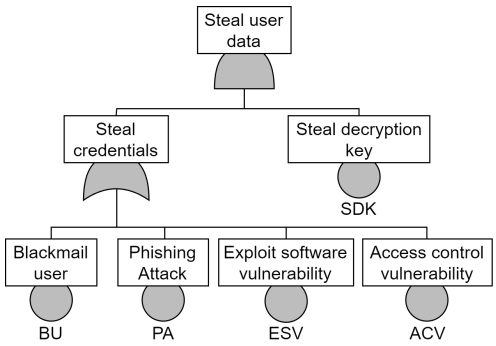


FIGURE 1.2: Attack tree depicting a scenario where the attacker aims to steal user data.

The hierarchical structure of an attack tree models the root of the tree as the attacker's primary goal. The tree branches represent different methods the attacker could take to achieve their primary goal. The leaves of these branches

are basic attack steps (BASs), which cannot be further refined into finer sub-goals. To exemplify this, consider Figure 1.2, which we will build on throughout the introduction, where the attacker aims to steal the defender's user data. This attack tree includes `AND` gates, activated when all of its children are enabled, and `OR` gates, activated when only a single child is enabled.

To obtain the user's data, the attacker must obtain both the defender's credentials and the decryption key. Stealing the decryption key (SDK) is challenging, as the defender stores it offline. Nevertheless, this step must be part of any successful attack. Subsequently, the credentials can be stolen in four different ways: blackmailing the user into handing over the password (BU), conducting a phishing attack where the user is tricked into revealing the credentials (PA), exploiting a software vulnerability to gain unauthorized access (ESV), or leveraging access control vulnerabilities (ACV) to accessing sensitive functionality.

1.1 Attack-Defense Trees

One of the limitations of attack trees is that they do not account for the countermeasures implemented to prevent an attack. For this reason, attack-defense trees (ADTs) were introduced by Kordy *et al.* [9] as an extension of regular ATs to model the attacks on a system concurrently and the defenses to block those attacks. Since their inception, ADTs have been effectively applied in the analysis of cyber-physical systems [10, 11], ATMs [12] and RFID-managed warehouses [13], highlighting their versatility across security-critical domains.

The main advantage attack-defense trees have over attack trees is their ability to model defenses. Each BAS has a binary activation value, where activated BASs are part of an attack and deactivated BASs are not. The defenses work by deactivating the attack nodes

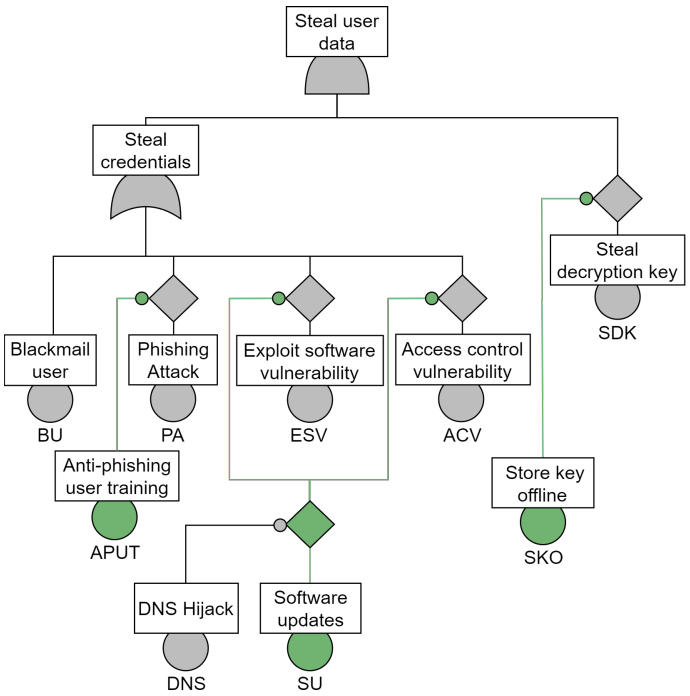


FIGURE 1.3: Attack-defense tree where the attacker aims to steal user data, and the defender can prevent attack nodes from being propagated.

they are associated with, thereby disabling them.

Figure 1.3 extends Figure 1.2 by adding counter-attack nodes. The defender can prevent phishing attacks through anti-phishing user training ($APUT$). Regular software updates (SU) prevent both ESV and ACV, resulting in a dynamic acyclic graph (DAG) structure. This is inspired by real-life attack trees, which often contain nodes that activate multiple parents [8, 14]. An interesting attack is DNS Hijack (DNS) which does not directly contribute to reaching the top node but disables the SU defense. As illustrated later, DNS can re-enable ESV and ACV. Lastly, blackmailing the user (BU) has no countermeasure, but the high attacker cost balances this.

The basic defense steps (BDS) $APUT$, SU, SKO are highlighted in green for better visualization. A defense node meets an attack node at an `INH` (inhibition) gate. This gate has exactly two children and is activated only when the attacking child is activated, but the counter-attack is not. Note that the counter-attack type is always the opposite of the attack type: DNS (an attack node) counters SU (a defense node); SKO (a defense node) counters SDK (an attack node). For clarity, the edge leading to the counter-attack child of `INH` gates is marked with a small circle.

1.2 Quantitative Analysis on Attack Trees

The structure of attack trees enables the quantitative analysis of attack metrics [15]. BASs are assigned attribute values representing measurable aspects of the attack scenario, such as the minimum cost or time required for an attack. Since an attack is composed of activated BASs, determining the impact of an attack involves propagating the attribute values to the root node through the intermediary gates.

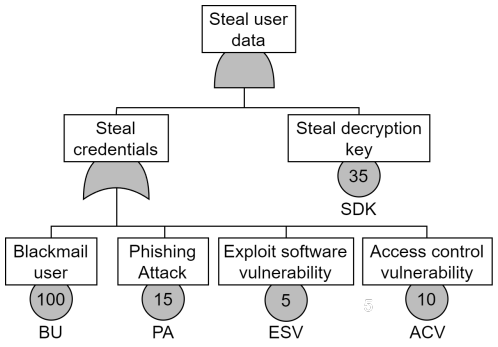


FIGURE 1.4: Attack tree annotated with cost values, depicting a scenario where the attacker aims to steal user data.

Figure 1.4 incorporates cost values into the BASs of the tree in Figure 1.2. In this scenario, the attack with the minimum cost is computed from the bottom to the top as follows. The *"Steal credentials"* gate is an `OR` gate, which requires the activation of only one child. Since the attacker seeks to minimize the attack cost, they should pick the BAS with the minimum cost, ESV. The *"Steal user data"* root is an `AND` node, requiring the activation of both the "*Steal credentials*" node and the SDK basic attack step, meaning their costs must be added together. This results in the minimum cost attack being $\{ESV, SDK\}$, with a total cost of 40.

1.3 Quantitative Analysis on Attack-Defense Trees

In the current state of the attack-defense trees research, although there are two actors in an attack scenario (i.e., an attacker and a defender), only the attacker's actions are annotated with quantifiable attribute values [16, 17]. This approach fails to fully capture reality, as the defender, like the attacker, usually has finite resources. For an illustration of how an ADT looks like with attributes for the attacker and defender, consider the tree in Figure 1.5, which extends Figure 1.3 by incorporating defender cost values. In this

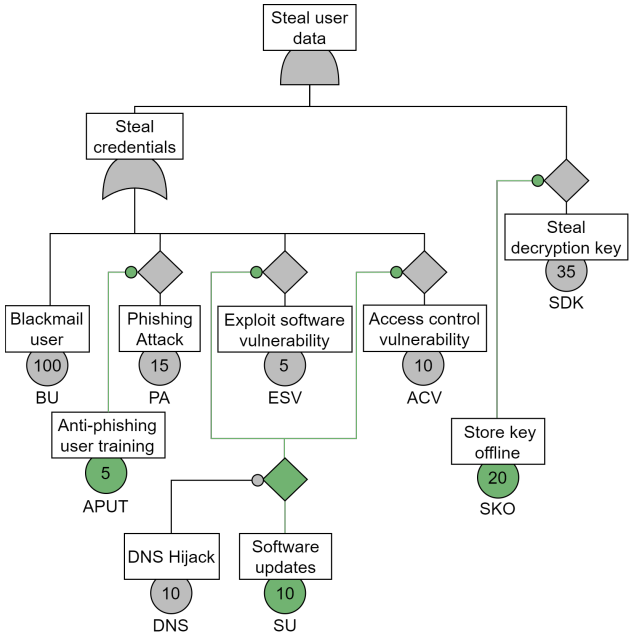


FIGURE 1.5: Attack-defense tree annotated with cost values where the attacker aims to steal user data, and the defender can prevent attack nodes from being propagated.

representation, both the BASs and BDSs are annotated with cost values, and each party aims to minimize their own cost.

1.4 Research Goal

The defender aims to select the most optimal defenses, typically those with a lower defense cost, while simultaneously making the situation more difficult for the attacker (e.g., maximizing the attacker cost). Note that the defender's and attacker's goals are conflicting: the attacker seeks to minimize their cost, whereas the defender aims to maximize it.

A solution that simultaneously satisfies both of the defender's goals does not exist. Instead, we aim to find the set of solutions which provide an overview of **all** optimal defense and attack metric values, known as the Pareto front. This leads to our research goal, the mathematical formulation of which is deferred to Section 3.4.

R.G.: Find efficient algorithms that compute the Pareto front between the metric values of the defender's and attacker's attribute domains for tree-structured and DAG-structured attack-defense trees.

A fundamental concept in this work is the dynamic between the defender and the attacker: the defender fixes their defenses first, and the attacker responds to them. Defenders set the stage for the attacker's responses by initially establishing the defences. This sequence is crucial, as it mirrors real-world scenarios where the defenders of a system already have resources allocated before any attack occurs [18]. Additionally, when planning system defenses, it is often best to prepare for worst-case scenarios [19]: the defender has no information about the attacker's plans, while the attacker is fully aware of the defender's actions. By leveraging the Pareto front, defenders can make better-informed decisions to allocate resources efficiently. The following section explores calculating the Pareto front for the example depicted in Figure 1.5 to clarify this further.

Row	Defense	Defense cost	Min. size attacks	Min. attack cost
1.	$\varnothing$	0	$\{\{BU, SDK\}, \{PA, SDK\}, \{ESV, SDK\}, \{ACV, SDK\}\}$	40
2.	$\{APUT\}$	5	$\{\{BU, SDK\}, \{ESV, SDK\}, \{ACV, SDK\}\}$	40
3.	$\{SU\}$	10	$\{\{BU, SDK\}, \{PA, SDK\}, \{DNS, ESV, SDK\}\}$	50
4.	$\{SKO\}$	20	$\varnothing$	$+\infty$
5.	$\{APUT, SU\}$	15	$\{\{BU, SDK\}, \{DNS, ESV, SDK\}\}$	50
6.	$\{APUT, SKO\}$	25	$\varnothing$	$+\infty$
7.	$\{SU, SKO\}$	30	$\varnothing$	$+\infty$
8.	$\{APUT, SU, SKO\}$	35	$\varnothing$	$+\infty$

TABLE 1.1: Quantitative analysis of the attack-defense tree from Figure 1.5.

1.5 Cost-Cost Pareto Front

The computation results are presented in Table 1.1, with the corresponding Pareto front depicted in Figure 1.6. The "*Defense*" column enumerates all possible combinations of fixed defenses, while their respective defense costs are in the "*Defense cost*" column. For each fixed defense, a list of all successful minimum size attacks (or $\varnothing$ if no such attacks exist) is illustrated in the "*Min. size attacks*" column. Finally, the cost metric for each minimal-size attack is computed, with the lowest cost value presented in the "*Min. attack cost*" column.

The first interesting observation from this quantitative analysis is that when the defender enables SKO, all attacks on the system will fail. In this scenario, they do not have to activate additional defenses, as all attacks have already been prevented. Furthermore, when the defender enables SU, the attacker can neutralize this defense with DNS. Notably, DNS is only present in defense suites that contain SU, as DNS alone does not contribute to reaching the top goal.

To find the Pareto front from Table 1.1, the first step is to plot the "*Defense cost*" and "*Min. attack cost*" values on a graph, labelling the points to their corresponding "*Row*" indexes. This plot is illustrated in Figure 1.6. To determine the Pareto optimal solutions, we need to identify all the points where no other point has both a lower or equal defense cost and a higher attack cost. In this case, the rows $(1), (3), (4)$ form the Pareto Front.

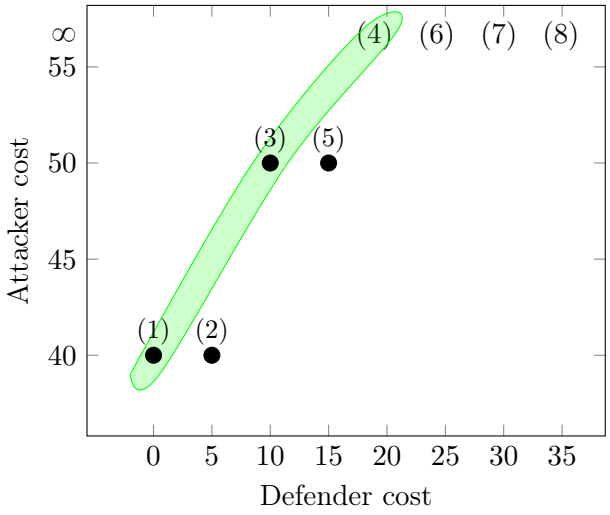


FIGURE 1.6: Pareto front of the attack-defense tree from Figure 1.5.

1.6 Contributions

Below, we outline our contributions. The subsequent chapters of this thesis are organized as follows. Chapter 2 introduces the basic concepts behind ADTs and reviews some previously used methodologies to analyze ATs and ADTs with only attacker's attributes. Chapter 3 presents the formal model behind ADTs and mathematically formulates the Pareto front problem. The `BU`, `Naive`, `BILP` and `BDD` algorithms are presented in the Chapters 4 through 7. In Chapter 8, we evaluate the algorithms on a wide array of randomly generated ADTs. Lastly, Chapter 9, we conclude the thesis and discuss potential avenues for future work.

The main **contributions** of this work are as follows:

1. General syntax and semantics for defining ADTs that support attacker and defender attribute domains.
2. Formal representation of metrics and the Pareto front between the attacker's and defender's attribute domains.
3. A bottom-up algorithm for finding the Pareto front of tree-structured ADTs.
4. A Biobjective Integer Linear Programming-based algorithm for finding the Pareto Front of DAGs using the minimum cost attribute domains.
5. A Binary Decision Diagram-based algorithm for finding the Pareto front of DAGs.
6. An extensive performance evaluation of the above algorithms on randomly generated ADTs.

Our experiments indicate that the bottom-up algorithm is the fastest for computing the Pareto front in tree-structured ADTs, whereas the BDD-based approach is the most efficient for DAG-structured ADTs.

Chapter 2

Background & related work

This chapter describes the literature advancements on concepts that are relevant to the quantitative analysis of attack-defense trees and the computation of the Pareto Front.

2.1 Attack Trees

Popularized by Schneier [2] and later formalized by Mauw and Oostdijk [15], attack trees provide a hierarchical framework for modelling the threats and vulnerabilities of a system. System owners can make informed decisions to prevent these attacks by identifying and categorizing successful attacks of a minimal size. In an attack tree, the attacker's primary objective is represented by the root, while the branches model various methods to achieve this goal. The leaf nodes, or basic actions, represent the fundamental actions of the attacker and cannot be further decomposed. Attack trees were initially inspired by fault trees, which were used for safety analysis of nuclear research in 1980 [20].

In Arnold *et al.* [21], the authors develop a framework to analyze the success probability of an attack as time progresses. This framework allows for the representation of the sequence of attack steps and the timing between them. Each leaf node in this framework is annotated with a probability distribution representing the time required for the basic attack step to be successful. These distributions are then propagated up the tree to derive a probability distribution for the entire system.

Despite the term *"tree"* in their name, attack trees might not necessarily have a tree-like structure. Generally, ATs can have two kinds of forms. When each node of the AT has a single parent, the AT maintains a tree-like structure, as illustrated in Figure 1.2. However, if a node has multiple parents (e.g., in Figure 1.3, the `INH` node of software updates has two parents), then the AT has a DAG-structure.

2.1.1 Attack-Defense Trees

In attack trees, the intermediate nodes are marked with gates that indicate the activation pattern of their children. In the structure of attack trees introduced by Schneier [2], only `AND` and `OR` gates are incorporated. However, this model only allows for representing basic situations where the dependencies between nodes or counter-attacks are not considered. Various extensions to this basic model have been developed to address this limitation, which help replicating more intricate scenarios.

One notable extension in this regard is attack-defense trees, a concept introduced by Kordy *et al.* [9], which allows for defense modelling through *counter-attack* gates. Before this work, there had already been defined ideologies of attack-defense trees, but they were

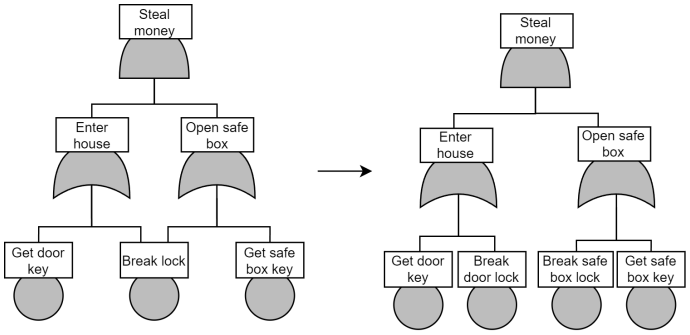


FIGURE 2.1: Transformation from DAG-structured ADT with separate instances to tree-structured ADT.

tailored to more specific use cases. For instance, in Bistarelli *et al.* [22], defensive actions are only possible at the leaf level. Similarly, in Roy *et al.* [23], the defender can only perform counter-attacks at the leaf level but cannot have higher-level goals modelled in the tree. In their paper, Kordy *et al.* [9] provided a more general concept of attack-defense trees, where attackers and defenders have equal capabilities, and counter-attacks can be modelled at intermediate nodes, including the root node. This approach offers a more comprehensive overview of the security aspects of a system.

In Arias *et al.* [24], the authors analyze ADTs in a novel way by treating these trees as an extension of asynchronous multi-agent systems. Each node in the tree is treated as an agent that can act asynchronously. The transition functions of these nodes are then equipped with attributes. Finally, the quantitative results from the generated automata are verified with state-of-the-art model checkers such as `UPPAAL` and `Imitator`.

The concept of attack defense trees is directly relevant to this research, providing a theoretical foundation for adding attributes for the defender. As mentioned in Section 1.3, we will introduce an inhibition gate to represent counter-attacks in the mathematical definition of attack defense-trees. Moreover, the semantics of ADTs will be extended with a separate attribute domain for the defender.

2.1.2 Directed Acyclic Graphs

When the tree is a directed acyclic graph (DAG), nodes can become the children of multiple intermediate nodes. In Bossuat and Kordy [25], the authors distinguish between two kinds of nodes that are shared between multiple parents. Firstly, if a node has multiple parents and its activation is propagated only to a single parent, the ancestors share different instances of the goal. This interpretation is straightforward, as each node activation can be treated separately, transforming the DAG into a tree-structured AT. For example, in Figure 2.1, the attack "*Break lock*" needs to be executed once to break the door lock and enter the house and once more to break the safe box to open it. As a result, this node can rewritten with the actions "*Break door lock*" and "*Break safe box lock*". This conversion represents the situation more accurately, as to reach the top `AND` gate, the attacker must perform "*Break lock*" twice.

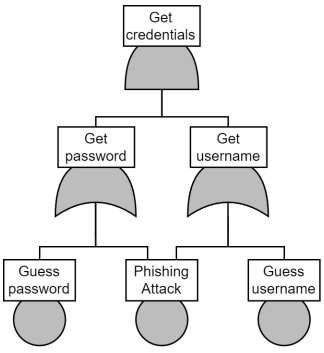


FIGURE 2.2: DAG tree with a shared instance of the node "*Phishing Attack*"

However, when the activation of a node is propagated to all its parents, the parents share the same instance of the goal. This scenario can be seen in Figure 2.2, where a single *"Phishing Attack"* retrieves both the password and the username simultaneously. Unlike Figure 2.1, this cannot be simplified in a tree-structured attack tree. In this case, the BU algorithm will not yield accurate results since it will propagate the attribute value of *"Phishing Attack"* multiple times instead of only once. Throughout the paper, when we specify that a tree has a DAG-structure, we refer to this latter case, where parents share the same instance of a node.

2.2 Quantitative Analysis

The semantic analysis of an attack tree, which identifies potential attacks and exposes vulnerabilities, is frequently combined with a quantitative analysis of the modelled scenario. This analysis is typically achieved through attributes representing quantifiable metrics or proprieties associated with nodes in the tree [15]. These attributes enable the computation of *metrics* such as the minimum time [2], required skill, maximum damage [15], cost [26] and probability [27] of performing an attack. These metrics can also be intertwined, creating dependencies, such as the minimum time of an attack given a maximum budget or skill level [28, 29]. Although the objective is to compute these metrics as efficiently as possible, the problem is generally NP-hard [26]. Fortunately, specific properties of an attack tree enable efficient methodologies to be applied.

A bottom-up (BU) approach is the most efficient one, propagating attribute values from the leaf nodes to the root node [15]. This operation is performed linearly in the number of nodes and edges in the tree, making it a fast algorithm. Another possible approach, though not frequently used due to its exponential complexity, involves first computing a complete attack suite and then selecting the attack with the desired properties [25]. This is an area of ongoing research, where some authors are building algorithms specialized for single metrics [30] while others aim to keep the algorithms as generic as possible [26].

2.2.1 Semirings

Generally, a wide array of AT metrics can be described as a semiring with three components: a set where the BASs take their value and two or more operations satisfying certain axioms [31]. In the context of AT metrics, this algebraic structure is called an *attribute domain*. For example, the attribute domain of the minimum cost metric is defined as $(\mathbb{R}_{\geqslant 0}, \min, +)$, where the value of a BAS is in $\mathbb{R}_{\geqslant 0}$, the disjunctive operation for the `OR` gate is min, and the conjunctive operation for `AND` is $+$. Unfortunately, there is no consensus in the literature on defining a specific metric, leading to possible inconsistencies between the works [31].

A vital aspect of semirings is that metrics defined on semirings attribute domains can be computed linearly via a bottom-up algorithm on the number of tree nodes and edges. This enables metrics to be calculated in a more generalizable way [15].

2.2.2 Quantitative Analysis on Tree-Structured ATs

The formalism of attack-defense trees can be extended with different classes of semantics: *propositional, multi set and equational*, each suited for different use case scenarios depending on what is considered to be an attack [32]. To analyze attribute domains, a framework based on propositional semantics was developed by Kordy *et al.* [32], which was later extended to support DAGs using multi-set semantics [16].

A bottom-up method can compute any metric linearly using *propositional* semantics when the attack tree has a tree-like structure [26]. Moreover, in cases where a single tree node represents separate instances (e.g., as shown in Figure 2.1), these can be separated into distinct nodes with unique labels, transforming the tree to a tree-structured form. In [25], the authors adopt this approach, replacing the labels with pairs in $\mathcal{G} \times \Gamma$, where $\mathcal{G}$ is a typed set of goals containing basic events, and Γ is a finite set of indices for distinguishing different instances of the same node. For example, in Figure 2.2 if we were to use the approach of Bossuat and Kordy [25] to represent two different instances for the goal ‘‘`Break door`’’ $\in \mathcal{G}$, the nodes would be labeled (‘‘`Break door`’’, 1) and (‘‘`Break door`’’, 2).

Therefore, to compute the Pareto front, we will first consider tree-structured ADTs and create a bottom-up algorithm that concurrently computes the metric values for both the defender and the attacker. Based on the findings from Mauw and Oostdijk [15], we expect this algorithm to work for any pair of metrics defined on semirings.

2.2.3 Quantitative Analysis on DAG-structured ATs

A classic result from the literature is that the bottom-up algorithm does not work on a DAG-structured tree (where a node has multiple parents that all share the same instance of that node) [16]. However, recent developments have found this is not always the case [26]. Let $(V, \oplus, \otimes)$ be an attribute domain. If the $\otimes$ operation absorbs $\oplus$, then even if the metric value of a node is propagated multiple times, the final result does not change.

To illustrate this concept, consider the tree in Figure 2.2 and the minimum skill domain $(\mathbb{R}_{\geqslant 0}, \min, \max)$. Formally, max absorbs min because for $x, y \in \mathbb{R}_{\geqslant 0}$, $\max(x, (\min(x, y)) = x$. Even if the skill value of *“Phishing Attack”* is propagated two times through the nodes *“Get password”* and *“Get username”*, applying max at the root `AND` node only retains the value of *“Phishing Attack”* once despite multiple propagations. On the other hand, if we were to apply $+$ at the root `AND` node as for the $(\mathbb{R}_{\geqslant 0}, \min, +)$ domain, the skill value from *“Phishing Attack”* would be doubled, as $+$ does not absorb min.

When the conjunctive operation $\otimes$ is not idempotent, various techniques other than the bottom-up have been explored over the years. Most of these strategies work by reformulating the issue into other mathematical and computer science domains such as Bayesian networks, game theory, satisfiability problems, or constraint optimization problems and addressing the problem within those contexts. For this reason, solving DAGs remains an open problem for attack trees, with a wide variety of proposed solutions. Below, we highlight the ones considered in our research, though this is not a comprehensive list of all possible approaches.

C-BU algorithm

One approach is to enhance the standard bottom-up algorithm by incorporating the concept of *necessary clones*. A clone is a node that has multiple parents. A clone is deemed *necessary* if it appears in all successful minimum attacks; otherwise, it is considered optional. The C-BU algorithm replaces the initial assignment value of clones with the neutral element of the domain so that the clones are effectively ignored during the BU procedure. For example, in the minimum cost attribute domain $(\mathbb{R}_{\geqslant 0}, \min, +)$, the neutral element for min is ∞, as for any $a \in \mathbb{R}$. $\min(a, \infty) = a$ [16]. Similarly, the neutral element for $+$ is 0.

At the end of the computation, the assignment values of necessary clones are reintroduced to ensure they are counted precisely once. Since the standard BU procedure is repeated for each subset of optional clones, the worst-case complexity of this algorithm is $\mathcal{O}(n2^k)$, where n is the number of nodes, and k the number of optional nodes.

Integer Linear Programming

Another approach for computing metrics involves using constraint solvers to determine whether a set of basic actions satisfies a given set of constraints in an attack tree [33]. Generally, an Integer Linear Programming (ILP) problem has the following form, where $\vec{c} \in \mathbb{R}^n$ is a vector of coefficients, $\vec{y}$ is a vector of integer variables and $A \in \mathbb{R}^{m \times n}$ represents the constraints [29].

$$\text{minimize } \vec{c} \cdot \vec{y} \text{ subject to } A \cdot \vec{y} \leqslant 0$$

ILP techniques have been effectively used to considerably decrease the computation time of dynamic attack trees when analyzing the minimum time metric [34]. More specifically, a new time assignment function f_v is created, which assigns a completion time to each tree node. When we consider the set of all time assignments to be $\mathcal{F}_T$, the optimization problem for a tree T becomes $\min_{f \in \mathcal{F}_T} f_{R_T}$. Since the constraints used in the definition of f are not linear, auxiliary integer variables are used to transform the AND, OR, and SAND gates into a set of linear constraints using standard integer programming techniques [35].

When multiple objectives need to be optimized simultaneously, such as finding both the cost and damage of a successful attack, this becomes a Multi-Objective Integer Programming task (MOIP), which generates a Pareto front [29]. Since dedicated solvers exist for such tasks, this approach focuses on translating the metrics problem into a real-world optimization problem with a linear set of constraints [36]. This solution can be particularly beneficial when the tree has a DAG structure, which may result in faster computation times compared to the C-BU algorithm.

Binary Decision Diagrams

Lopuhaä-Zwakenberg *et al.* [26] translate the AT to a boolean function, which is then used to create a Binary Decision Diagram. For instance, the boolean function behind the DAG in Figure 2.2 is $(\texttt{GuessP} \vee \texttt{PhishA}) \wedge (\texttt{GuessU} \vee \texttt{PhishA})$. This boolean function can be efficiently and compactly represented as a BDD and standard BDD techniques can be used to compute metric values.

However, for this to work, the $\oplus$ operation of the attribute domain needs to be absorbing and idempotent, where idempotency is defined as follows: for all $d \in D$, $d \oplus d = d$. In plain terms, if an idempotent operation is applied multiple times, it should have the same effect as applying it once.

Other approaches

Alternative methods have been developed to accurately handle the quantitative evaluation of DAGs and are usually used when the BU algorithm is incompatible [3]. Some of them work by first translating the tree into another formal object and then carrying out computations on this object [28]; for example, encoding SAT problems as generalized stochastic Petri nets [37] or even game-theory approaches [38] where the Nash equilibrium between the defender and attacker is found.

The primary limitation of these approaches is that they do not model a separate attribute domain for the defender. Instead, they only measure the attack metrics when a subset of the defenses are activated, without assigning any quantifiable attributes to the defenses.

2.2.4 Quantitative Analysis on Attack-Defense Trees

The quantitative aspects of attack defense trees extend those of regular attack trees by adding to the attribute domain the concept of defensive nodes [32]. As mentioned in section 2.2.1, attribute domains for attack trees are formed of three parts: a domain, an operation for `AND` gates, and one for `OR` gates. However, when the actions of both the attacker and defender are modelled, then the `AND`, `OR`, and counter-attack gates can represent either a defensive or offensive action, leading to an attribute domain composed of 7 elements. The concepts described for attack trees remain applicable to this extended attribute domain.

2.3 Pareto Front

Most analytical methods optimize one parameter at a time, such as the cost or time of an attack. However, this approach might not accurately represent complex real-world scenarios where parameters can interact (e.g., the maximum damage of an attack, given a fixed cost [29]), potentially leading to potentially sub-optimal solutions. To analyze multiple parameters simultaneously, the leaf nodes need to be annotated with multiple values. This creates a multi-optimization problem, as there is no single solution anymore, but a set of Pareto efficient solutions called the *Pareto Front*, where one solution is not dominated by another in a given ordering relation [17]. For instance, the main intuition behind this ordering, assuming the Pareto front between the attacker's damage and cost, is that if the attacker has two strategies with the same damage, but one has a lower cost, they have no incentive to choose the higher-cost strategy.

Instead of solving a single-objective ILP, Lopuhaä-Zwakenberg and Stoelinga [29] focus on multi-objective ILP to find the Pareto Front between the attacker's costs and damage values. Often, as the attacker can spend a higher amount, the damage inflicted on the system also increases.

Another possible approach to finding the Pareto front is through a regular bottom-up method by combining the attribute domain of each parameter into a single Pareto attribute domain suitable for analysis [17, 39]. Here, each basic assignment β_{α_i} for the metric α_i is combined into a singleton $\{(\beta_{\alpha_1}(b), \ldots, \beta_{\alpha_m}(b))\}$, which will be used to redefine new $\hat{\oplus}$ and $\hat{\otimes}$ operations. The authors prove that a Pareto attribute domain formed in this manner is a commutative idempotent semi-ring, which can then be analyzed using the BU procedure. If the tree can be transformed into a tree structure, a single bottom-up procedure can find the solution using this approach. Furthermore, the method is effective even if the tree is a DAG as long as the attribute domains of the parameters are absorbing, leading to an exponential complexity in the number of distinct nodes.

It is important to note that although Pareto Fronts have previously been explored for ATs, these studies focused on the trade-offs between multiple attacker's metrics. In our work, we consider the trade-off between the defender and attacker metrics.

Chapter 3

Attack-Defense Trees

The main objective of this section is to introduce the notation used for attack-defense trees with offensive and defensive attributes and formalize the thesis' goal.

Symbol	Description	Page
$T = (N, E)$	Attack-defense tree	13
T_v	Sub-tree rooted at node v	13
R_T	Root of T	13
$ch(v)$	Children of node v	13
$\gamma(v)$	Gate type of v	13
$\tau(v)$	Node type of v	13
$\overline{\tau(v)}$	Opposite node type of v	13
$\mathbb{B}$	$\{0, 1\}$	14
$\mathcal{A}$	Basic attack steps of T	14
$\mathcal{D}$	Basic defense steps of T	14
$\vec{\delta}, \vec{\alpha}$	Defense, attack vectors	14
$\mathbb{D}_D, \mathbb{D}_A$	Attribute domains of defender, attacker	15
$\beta_D(d)$	Attribute assignment of defense d	15
$\beta_A(a)$	Attribute assignment of attack a	15
$\oplus$	Abstract disjunctive operation	15
$\oplus$	Abstract conjunctive operation	15
$\rho(\vec{\delta})$	Attacker's response based on $\vec{\delta}$	16
$\mathcal{S}$	Strategies set	16
$f_T(\vec{\delta}, \vec{\alpha}, v)$	Structure function of T	16
$\hat{\beta}(\vec{\delta}, \rho(\vec{\alpha}))$	Metric value of the strategy $(\vec{\delta}, \rho(\vec{\alpha}))$	15
$\preceq_D, \preceq_A$	Partial orderings for the metrics	17
$\sqsubseteq$	Ordering relation for the Pareto front	17
$PF_S(T)$	Pareto front of T based on semantics	17
$\underline{\min}_{\sqsubseteq} \hat{\beta}(\mathcal{S})$	Formal Pareto Front problem	17

TABLE 3.1: Notation used throughout Chapter 3.

3.1 Syntax

This section reviews the formalisms of attack-defense trees, starting with their definition.

Definition 1. An attack-defense tree is defined as a quadruple $T = (N, E, \gamma, \tau)$, where (N, E) forms a rooted acyclic graph, and each node $v \in N$ has a gate type $\gamma(x) \in \{\texttt{OR}, \texttt{AND}, \texttt{BS}, \texttt{INH}\}$ and belongs to an attacker or a defender: $\tau(v) \in \{\texttt{A}, \texttt{D}\}$.

Moreover, T satisfies the following constraints for a node $v \in N$:

- **Leaf-BS**: $\gamma(v) = \texttt{BS}$ if and only if v is a leaf of (N, E).
- **INH-Children**: $\gamma(v) = \texttt{INH}$ if and only if v has exactly two children v_a and v_c representing the attack and counter-attack respectively. Using the notation $\overline{\tau(v)}$ to denote the opposite τ value of node v, $\tau(v_a) = \tau(v)$ and $\tau(v_c) = \overline{\tau(v)}$.
- **OR-AND-Children**: if $\gamma(v) \in \{\texttt{OR}, \texttt{AND}\}$, then for all children w of v, $\tau(w) = \tau(v)$.

The children of a node $v \in N$ are defined by the function $ch(v) = \{w \mid (v, w) \in E\}$. The tree has a unique root labelled R_T, where $\forall v \in N. R_T \notin ch(v)$. The sub-tree rooted at node v is denoted as T_v. Additionally, the notation $v = \texttt{OR}(v_1, \ldots, v_n)$ is used when $\gamma(v) = \texttt{OR}$ and $ch(v) = (v_1, \ldots, v_n)$, and similarly for $\texttt{AND}$, $\texttt{INH}$ nodes.

Despite their name, attack-defense trees do not necessarily have a tree-like structure. For an ADT to be an actual *tree*, a node must not have multiple parents, i.e. $\forall u, v \in N, ch(u) \cap ch(v) = \varnothing$. When this property is not satisfied, the ADT has a DAG structure.

The set of all basic steps in T is denoted by B_T, or simply B if there is no confusion. For simplicity, the set of all basic attack steps is denoted by $\mathcal{A}$, when for a node $v \in N$, $\gamma(v) = \texttt{BS}$ and $\tau(v) = \texttt{A}$. Similarly, the set of all basic defense steps is denoted by $\mathcal{D}$, when $\gamma(v) = \texttt{BS}$ and $\tau(v) = \texttt{D}$. These two sets are disjoint, meaning $\mathcal{A} \cap \mathcal{D} = \varnothing$, and their union represents the set of all basic events, i.e., $\mathcal{A} \cup \mathcal{D} = B$.

Let $\mathbb{B} = \{0, 1\}$ to represent the set of booleans, with the logical operators $\wedge$ (AND) and $\vee$ (OR). For an attack-defense tree T, let $\vec{\delta} \in \mathbb{B}^{\mathcal{D}}$ represent the defense vector where $\delta_d = 1$ when the defense $d \in \mathcal{D}$ is activated, and $\delta_d = 0$ if it is not. Analogously, $\vec{\alpha} \in \mathbb{B}^{\mathcal{A}}$ models the attack vector.

Example 1. To illustrate the concepts presented so far and the ones that will follow, let us consider a practical example and expand upon it as the chapter progresses. Figure 3.1 shows a tree-structured ADT annotated with numbers representing the cost. In this illustration, the set of all attacks is $\mathcal{A} = \{a_1, a_2, a_3\}$, and the set of defenses is $\mathcal{D} = \{d_1, d_2\}$. If the attacker activates a_2 and a_3, but not a_1, this forms the attack vector $\vec{\alpha} = (0, 1, 1)$. Similarly, a defensive vector where only d_1 is activated is represented by $\vec{\delta} = (1, 0)$.

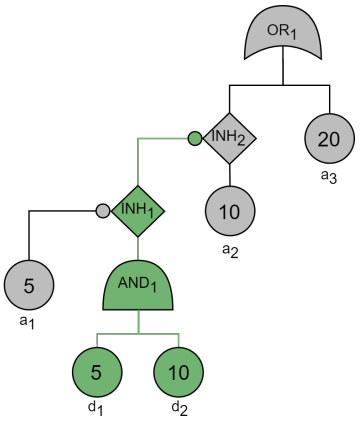


FIGURE 3.1: Tree-structured ADT annotated with offensive and defensive costs.

3.2 Attributes

Attack-defense trees are commonly used in quantitative analysis to compute security metrics, which indicate the performance of a system. Traditionally in attack trees, all the basic attack steps hold values assigned by a function β. Furthermore, a function $\hat{\beta}$ is used to determine the metric value for an attack. These values are then propagated up the tree towards its root, using the corresponding operations at the intermediary nodes. Table 3.2 contains some of the frequently used attribute domains.

METRIC	V	$\oplus$	$\otimes$
min cost	$\mathbb{R}_{\geqslant 0}$	min	$+$
min time (sequential)	$\mathbb{R}_{\geqslant 0}$	min	$+$
min time (parallel)	$\mathbb{R}_{\geqslant 0}$	min	max
min skill	$\mathbb{R}_{\geqslant 0}$	min	max
max challenge	$\mathbb{R}_{\geqslant 0}$	max	max
max damage	$\mathbb{R}_{\geqslant 0}$	max	$+$
discrete prob.	$[0, 1]$	max	$\cdot$
continuous prob.	$\mathbb{R} \to [0, 1]$	max	$\cdot$

TABLE 3.2: Semiring attribute domains

To enable computation, metrics can generally be described as attribute domains of the form $(V, \oplus, \otimes)$, where the *disjunctive* operation $\oplus$ and *conjunctive* operation $\otimes$ are associative and commutative. This algebraic structure is called a semiring if $\otimes$ distributes over $\oplus$. These propreties are outlined below:

- **Associativity.** $\forall a, b, c \in V.\ a \oplus (b \oplus c) = (a \oplus b) \oplus c$
- **Commutativity.** $\forall a, b, \in V.\ a \oplus b = b \oplus a$
- **Distributivity.** $\forall a, b, c \in V.\ a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

A semiring may also posses additional proprieties such as *idempotency* and *absorbtion*, though these are not required for the structure to qualify as a semiring. More specifically, a semiring is *idempotent* if the operation $\oplus$ is idempotent and *absorbing* if $\otimes$ absorbs $\oplus$:

- **Idempotency of $\oplus$.** $\forall a \in V.\ a \oplus a = a$
- **Absorbtion of $\oplus$.** $\forall a, b \in V.\ a \oplus (a \otimes c) = a$

Since, in this thesis the defender and attacker have separate attributes, we combine these two into an attribute pair defined as follows:

Definition 2. An attribute pair for an attack-defense tree T is a tuple $(\mathbb{D}_D, \mathbb{D}_A, \beta_D, \beta_A)$, where $\mathbb{D}_D\colon (V_D, \oplus_D, \otimes_D)$ and $\mathbb{D}_A\colon (V_A, \oplus_A, \otimes_A)$ are semirings. Each domain has an associated basic assignment function: $\beta_D\colon \mathcal{D} \to V_D$ and $\beta_A\colon \mathcal{A} \to V_A$ for the basic steps of the defender and attacker respectively.

The defender and attacker attribute domains are described by $\mathbb{D}_D$ and $\mathbb{D}_A$, respectively. β_D assigns an attribute value from V_D to each basic defense step in $\mathcal{D}$, while β_A does so to each basic attack step in $\mathcal{A}$. Since the value of the defender's metric lies in V_D, while that of the attacker is in V_A, their attribute pair will naturally have values in $V_D \times V_A$. The following are defined:

Definition 3. The metric value of a defense vector $\vec{\delta}$ is given by:

$$\hat{\beta}_D\colon \mathbb{B}^{\mathcal{D}} \to V_D \text{ with } \hat{\beta}_D(\vec{\delta}) = \bigotimes_{d \in \vec{\delta}}{}_D\, \delta_d \beta_D(d)$$

while the metric value of an attack vector $\vec{\alpha}$ is given by:

$$\hat{\beta}_A\colon \mathbb{B}^{\mathcal{A}} \to V_A \text{ with } \hat{\beta}_A(\vec{\alpha}) = \bigotimes_{a \in \vec{\alpha}}{}_A\, \alpha_a \beta_A(a)$$

Lastly, the metric value of a pair of defense and attack vectors is given by:

$$\hat{\beta}\colon \mathbb{B}^{\mathcal{D}} \times \mathbb{B}^{\mathcal{A}} \to V_D \times V_A \text{ with } \hat{\beta}\big((\vec{\delta}, \vec{\alpha})\big) = \big(\hat{\beta}_D(\vec{\delta}), \hat{\beta}_A(\vec{\alpha})\big)$$

Continuing Example 1. For more clarity, let us note a defense vector using the names of the activated nodes while omitting the disabled ones. Analogously for an attack vector. Let $\big(\{d_1, d_2\}, \{a_1, a_2\}\big)$ be a pair of defense and attack vectors. Since we are working with the minimal cost domain, $\mathbb{D}_A = \mathbb{D}_D = (\mathbb{R}_{\geqslant 0}, \min, +)$. To determine the metric values of this vector pair, we apply the definition of $\hat{\beta}$:

$$\begin{aligned}
\hat{\beta}_D\big(\{d_1, d_2\}\big) &= 1 \times \beta_D(d_1) + 1 \times \beta_D(d_2) \\
&= 5 + 10 = 15 \\
\hat{\beta}_A\big(\{a_1, a_2\}\big) &= 1 \times \beta_A(a_1) + 1 \times \beta_A(a_2) + 0 \times \beta_A(a_3) \\
&= 5 + 10 + 0 = 15 \\
\hat{\beta}\big(\big(\{d_1, d_2\}, \{a_1, a_2\}\big)\big) &= (15, 15)
\end{aligned}$$

Having defined how a pair of defense and attack vectors is assigned a numerical value, we analyze which vector pairs form a strategy in the next section.

3.3 Semantics

The semantics of an attack-defense tree are defined by what constitutes an event within the tree and when it is considered successful. In ADTs, an event is represented by a pair of defense and attack vectors $(\vec{\delta}, \vec{\alpha})$.

Formalizing the idea that `AND` gates are activated when all their children are also activated, and similarly for `OR` and `INH` gates with their respective conditions, the structure function of T is defined as follows:

Definition 4. The structure function $f_T \colon \mathbb{B}^{\mathcal{D}} \times \mathbb{B}^{\mathcal{A}} \times N \to \mathbb{B}$, indicates whether $(\vec{\delta}, \vec{\alpha})$ reaches the node v:

$$f_T(\vec{\delta}, \vec{\alpha}, v) = \begin{cases} \alpha_v, & \text{if } v \in \mathcal{A} \\ \delta_v, & \text{if } v \in \mathcal{D} \\ \bigwedge_{w \in ch(v)} f_T(\vec{\delta}, \vec{\alpha}, w), & \text{if } \gamma(v) = \texttt{AND} \\ \bigvee_{w \in ch(v)} f_T(\vec{\delta}, \vec{\alpha}, w), & \text{if } \gamma(v) = \texttt{OR} \\ f_T(\vec{\delta}, \vec{\alpha}, v_{\tau(v)}) \wedge \neg f_T(\vec{\delta}, \vec{\alpha}, v_{\overline{\tau(v)}}) & \text{if } \gamma(v) = \texttt{INH} \end{cases}$$

Since the defender acts first, an optimal attacker's response will always depend on $\vec{\delta}$. This relationship can be modelled as follows:

Definition 5. An attacker's response to a defense is a function $\rho \colon \mathbb{B}^{\mathcal{D}} \to \mathbb{B}^{\mathcal{A}} \cup \{\bot\}$ which maps a defense vector to the most optimal attack vector from the perspective of the attacker, or $\bot$ if there is no optimal response.

Depending on the chosen pair of metrics, $\rho(\vec{\delta})$ is found by maximizing the damage or probability of an attack, minimizing the cost or time, or other criteria. However, the attacker might not always have an optimal response. In such cases, $\rho(\vec{\delta})$ has no solutions. It's important to distinguish $\vec{0}$ from $\bot$, as in some scenarios, not activating any attacks might represent the optimal response.

Definition 6. A pair of defense and attack vectors $(\vec{\delta}, \vec{\alpha})$ is considered a "strategy" of T if $\vec{\alpha} = \rho(\vec{\delta})$. Let $\mathcal{S}$ be the set of all strategies of T, where:

$$\mathcal{S} = \left\{ \left(\vec{\delta}, \rho(\vec{\delta})\right) \mid \vec{\delta} \in \mathbb{B}^{\mathcal{D}} \right\}$$

The definition of a successful strategy varies based on the chosen pair of metrics. For example, a strategy based on the minimum cost domain is successful when the top-level goal is reached, but this might not hold for the maximum cost domain. As detailed in Section 3.5, this thesis focuses on the cost-cost attribute domain, where a successful strategy is defined as follows:

Definition 7. A strategy $\left(\vec{\delta}, \rho(\vec{\delta})\right)$ is successful when: $f_T\left(\vec{\delta}, \rho(\vec{\delta}), R_T\right) = \begin{cases} 1 & \text{if } \tau(R_T) = \texttt{A} \\ 0 & \text{otherwise.} \end{cases}$

In this definition, note that if R_T is an attack node, the attacker aims to reach the top node. Conversely, if R_T is a defense node, the defender aims to prevent the attacker from reaching the top node.

Continuing Example 1. The defense and attack vectors $\left(\{d_1, d_2\}, \{a_1, a_2\}\right)$ form a strategy because $\{a_1, a_2\}$ minimizes the attack cost. Activating d_1 and d_2 also activates AND_1,

thereby enabling the defense for INH_1. The attacker can choose between a_1 and a_3. Although a_3 creates a successful attack, its cost is not minimal. Instead, the attacker can use a_1 to disable INH_1, regaining access to a_2, which can then be used to reach the root node. The combined costs of a_1 and a_2 result in a total cost of 15. Therefore, $\vec{\delta} = \{d_1, d_2\}$ and $\rho(\vec{\delta}) = \{a_1, a_2\}$. Note that choosing any $\vec{\alpha}$ other than $\{a_1, a_2\}$ does not form a strategy.

3.4 Pareto Front

The trade-off between the defender's and attacker's metric values can be analyzed via the Pareto front. A point on the Pareto front is called *Pareto optimal* if no other solution is better in all objectives. The notion of *better* is formally known as dominance, and to define it, a few prerequisite properties need to be established.

The primary focus of this thesis is on the attribute domains based on semirings. A partial order relation $\preceq$ for a semiring $(V, \oplus, \otimes)$ must satisfy the basic properties of a partial order:

- **Reflexivity**: for any $x \in V. x \preceq x$, meaning each element is comparable with itself.
- **Anti-simmetry**: for any $x, y \in V.$ if $x \preceq y$ and $y \preceq x$, then $x = y$, indicating that if two elements are mutually comparable, they are the same element.
- **Transitivity**: for any $x, y, z \in V.$ if $x \preceq y$ and $y \preceq z$, then $x \preceq z$.

Furthermore, $\preceq$ must be compatible with the $\oplus$ and $\otimes$ operations of the semiring:

- **Monotonicity** with respect to $\oplus$: for any $x, y, z \in V.$ if $x \preceq y$ then $x \oplus z \preceq y \oplus z$, meaning adding the same element on both sides of an inequality does not change its direction.
- **Monotonicity** with respect to $\otimes$: for any $x, y, z \in V.$ if $x \preceq y$ then $x \otimes z \preceq y \otimes z$, meaning multiplying both sides of an inequality does not change its direction.

The linear order relation $\preceq_A$ is defined for $\mathbb{D}_A$, and $\preceq_D$ for $\mathbb{D}_D$, where both must satisfy reflexivity, anti-symmetry, transitivity, and monotonicity for $\oplus$ and $\otimes$. Using these orders, it is possible to formulate when one strategy is better, i.e., dominates another:

Definition 8. Given two strategies $\in \mathcal{S}$, and their valuations (d_1, a_1) and (d_2, a_2), the pair (d_1, a_1) dominates (d_2, a_2) i.e., $(d_1, a_1) \sqsubseteq (d_2, a_2)$ when $d_1 \leqslant d_2$ and $a_1 \geqslant a_2$.

As mentioned at the beginning of the section, for a general poset $(X, \sqsubseteq)$, a point $x \in X$ is optimal if it is not dominated by any other point in X. The Pareto front is the set of all Pareto optimal points in X, i.e., $\underline{\min}_{\sqsubseteq} X = \{x \in X \mid \forall x' \in X.\ x' \neq x, x' \not\sqsubset x\}$.

With all the necessary mathematical prerequisites defined, we can formally state the problem statement of the thesis:

Research Goal. For an attack defense tree T, we aim to find $\underline{\min}_{\sqsubseteq} \hat{\beta}(\mathcal{S}) \subseteq V_D \times V_A$, denoted $\mathrm{PF}_S(T)$, where S stands for semantics.

3.5 Minimum Cost Pair

As a practical example, let us take the minimum cost metric from 3.2 and use it for the defender and the attacker. This means that both parties are interested in minimizing their own cost.

The remainder of the thesis will use this pair to enhance readability in the provided algorithms and proofs. Generally, the algorithms are translated back into their abstract form by substituting the min operation with $\oplus$ and + with $\otimes$, which should make them applicable for any attribute based on idempotent absorbing semirings. However, readers should exercise caution when doing so, as this translation has yet to be formally verified, being outside the scope of this thesis.

The minimum cost pair is defined by the attribute pair $(\mathbb{D}_D, \mathbb{D}_A, \beta_D, \beta_A, \preceq_D, \preceq_A)$ where $\mathbb{D}_D = \mathbb{D}_A = (\mathbb{R}_{\geqslant 0}, \min, +)$. These domains have $\preceq_D = \preceq_A = \leqslant$ as order relations, where $\leqslant$ is the natural ordering for $\mathbb{R}$. Furthermore, the domains are *idempotent* and *absorbing*.

Assuming that evaluation of the strategy $\big(\vec{\delta}, \rho(\vec{\alpha})\big)$ results in the point $(d, a) \in \mathbb{R} \times \mathbb{R}$, the natural language interpretation of this tuple is as follows:

Meaning of a value pair: For a node v with a strategy evaluation (d, a), when:

$\tau = \texttt{D}$ and the defender spends at least d, the node v is **activated** unless the attacker spends at least a.

$\tau = \texttt{A}$ and the defender spends at least d, the node v is **not activated** unless the attacker spends at least a.

We can now also define the cost of a defense and attack vector more precisely:

$$\hat{\beta}_A(\vec{\alpha}) = \sum_{a \in \vec{\alpha}} \alpha_a \beta_A(a)$$

$$\rho(\vec{\delta}) = \begin{cases} \underset{f_T(\vec{\delta},\vec{\alpha},R_T)=1}{\operatorname{argmin}} \ \hat{\beta}_A(\vec{\alpha}) & \text{if } \tau(R_T) = \texttt{A} \\ \underset{f_T(\vec{\delta},\vec{\alpha},R_T)=0}{\operatorname{argmin}} \ \hat{\beta}_A(\vec{\alpha}) & \text{if } \tau(R_T) = \texttt{D} \\ \bot & \text{if argmin does not exist} \end{cases}$$

$$\hat{\beta}_D(\vec{\delta}) = \sum_{d \in \vec{\delta}} \delta_d \beta_D(d)$$

$$\hat{\beta}_A(\rho(\vec{\delta})) = \begin{cases} \underset{f_T(\vec{\delta},\vec{\alpha},R_T)=1}{\min} \ \hat{\beta}_A(\vec{\alpha}) & \text{if } \tau(R_T) = \texttt{A} \\ \underset{f_T(\vec{\delta},\vec{\alpha},R_T)=0}{\min} \ \hat{\beta}_A(\vec{\alpha}) & \text{if } \tau(R_T) = \texttt{D} \\ \infty & \text{if } \rho(\vec{\delta}) = \bot \end{cases}$$

Continuing Example 1. Another strategy for the tree in Ex. 1 is $\big(\{d_1\}, \{a_2\}\big)$, where $\vec{\delta} = \{d_1\}$ and $\rho(\vec{\delta}) = \{a_2\}$. Note that enabling only d_1 without d_2 does not activate AND_1. Thus, the defender is practically better off doing nothing. Nevertheless, this still forms a strategy, though not one which results in a Pareto optimal point. In this specific case, all the $\vec{\alpha}$ which satisfy $f_T(\vec{\delta}, \vec{\alpha}, R_T) = 1$ are $\{a_2\}$, $\{a_3\}$, $\{a_2, a_1\}$, $\{a_3, a_1\}$, $\{a_1, a_2, a_3\}$. From these, $\{a_2\}$ minimizes the cost: $\hat{\beta}_A\big(\{a_2\}\big) = 10$, which makes it the optimal attacker's response.

Chapter 4

Tree-structured ADTs

This chapter explores attack-defense trees with a tree structure and a fast bottom-up approach to find the Pareto front between the defender's and attacker's cost. We will detail the reasoning behind how the algorithm handles different types of gates and nodes and end the chapter with a formal proof of the algorithm's correctness.

4.1 Bottom-up Algorithm

For metrics defined on semirings, a bottom-up algorithm can be used to compute the Pareto front linearly, in $|N| + |E|$ (number of nodes and edges) when the AT is tree-structured, meaning none of the nodes share multiple parents [15]. To adapt the existing bottom-up algorithm for ADTs, the following steps are performed for a node $v \in N$:

$\gamma(v)$	$\tau(v)$	Def. op $(\bigcirc_D)$	Att. op $(\bigcirc_A)$
AND	A	$\sum$	$\sum$
	D	$\sum$	min
OR	A	$\sum$	min
	D	$\sum$	$\sum$
INH	A	$\sum$	$\sum$
	D	$\sum$	min

TABLE 4.1: Operators applied in the bottom-up algorithm.

1. Compute the Pareto Front bottom-up for each $w \in ch(v)$.
2. Identify all possible combinations of points from the children's Pareto Fronts.
3. For each combination, apply the min or + operations across the defense and attack costs, depending on the values of $\gamma(v)$ and $\tau(v)$.
4. Discard the dominated points from the previous step.

In step 1, the bottom-up algorithm is recursively applied to each child of v. Due to this recursive nature, the algorithm will first complete for the leaf nodes, and the results will then be propagated up the tree towards the root node (hence the *bottom-up* naming). For step 2, the Cartesian product is used to find all possible ways to combine the children's Pareto fronts. Unfortunately, computing all combinations is unavoidable, as it is impossible to predict which points will be included in the Pareto front before evaluating all the value pairs in step 3.

The objective of step 3 is to transform a vector of value pairs into a single value pair (d, a) for each possible combination. Recall the meaning of a value pair from Section 3.5: *if the defender spends at least d, the node v is **activated/not activated** unless the attacker spends at least a.* With this in mind, *Table* 4.1 describes each operator applied, depending on the values of $\gamma(v)$ and $\tau(v)$:

- For a basic attack step, the defender has no node that can be activated, which leads to a defense cost of 0. The least amount the attacker needs to spend to activate v is $\beta_A(v)$. This leads to the P.F. $\{(0, \beta_A(v)\}$.
- For a basic defense step, the defender can spend either 0 or $\beta_D(v)$. When the defender and attacker spend at least 0 (i.e. do nothing), v remains activated. Similarly, if the defender spends at least $\beta_D(v)$, no optimal attack is possible, illustrated by an attacker cost of ∞. This leads to the P.F. $\{(0,0), (\beta_D(v), \infty)\}$.
- In the case of an `AND` node, a strategy is successful if and only if all of its child strategies are also successful. Although the defender's goal is to minimize the defense cost, this is not possible yet, since the defender has to act before the attacker and choose a defense for each child. In this case, the defender's cost will be minimized in step 4, where the non-optimal points are discarded. This intuition applies to the following gate types as well, so we will only describe the attacker's rationale:
  - If $\tau(v) =$ `A`, then to enable v, the attacker needs to activate all children, leading to a sum.
  - If $\tau(v) =$ `D`, then to disable v, the attacker only needs to disable a single child. Thus, the child with the minimum cost is picked.
- When it comes to an `OR` node, the attacker's intuition is opposite from that of an `AND` node.
  - If $\tau(v) =$ `A`, then to enable v, the attacker can pick the child with the minimum cost.
  - If $\tau(v) =$ `D`, then to disable v, the attacker needs to activate all children, leading to a sum.
- The `INH` gate has only two children v_D, v_A, with $\tau(v_D) =$ `D`, and $\tau(v_A) =$ `A`. By taking the Cartesian product of the Pareto fronts of v_D and v_A to find all possible strategies:
  - If $\tau(v) =$ `A`, then to enable v, the attacker needs to enable v_A **and** disable v_D. Thus, the attacker costs of v_A and v_D need to be summed.
  - If $\tau(v) =$ `D`, then to disable v, the attacker can either disable v_D **or** enable v_A. Thus, the attacker picks the option with a lower cost.

The output of step 3 is a set of value pairs. In step 4, we reduce the elements of this set to the Pareto Front by discarding all the dominated points according to Definition Def. 8. The complete algorithm is presented in Alg. 1, where `FindPF` represents step 4 specifically.

Example 2. Let us consider the tree in Ex. 1 to exemplify how the algorithm operates step by step. Since the root is an attack `OR` gate, according Table 4.1, we need to compute $\texttt{FindPF}(\{(\sum_{i=1}^{n} d_i, \min_{i=1}^{n} a_i) \mid (\vec{d}, \vec{a}) \in \times_{u \in ch(v)} \mathrm{BU}(T, u, \beta)\})$. In Table 4.2, we divide this goal into three sub-goals:

- 2^{nd} column (Algorithm step 2): Find the Cartesian product between the children's Pareto Fronts.
- 3^{rd} & 4^{th} column (Algorithm step 3): For each pair in 2^{nd} column, apply the appropriate operators to find the value pairs.
- 5^{th} column (Algorithm step 4): Reduce the values from the 4^{th} column to the Pareto Front.

Algorithm 1 Bottom-up

Input:
T: attack defense tree
v: node $v \in N$
β: assignment of nodes $\in N$
Output: Pareto frontier of the sub-tree rooted at v.

1: **procedure** BU(T, v, β)
2: $pts \leftarrow$ new array
3: **if** $v \in \mathcal{A}$ **then**
4: **return** $\{(0, \beta_A(v))\}$
5: **else if** $v \in \mathcal{D}$ **then**
6: **return** $\{(0,0), (\beta_D(v), \infty)\}$
7: **else**
8: $p \leftarrow \times_{u \in ch(v)} \text{BU}(T, u, \beta)$ ▷ Step 2
9: $pv \leftarrow \{(\bigcirc_{D\,i=1}^{n} d_i, \bigcirc_{A\,i=1}^{n} a_i) \mid (\vec{d}, \vec{a}) \in p\}$ ▷ Step 3
10: **return** FindPF(pv) ▷ Step 4

Input: pts: set of strategy values.
Output: Subset of non-dominated values from pts.

1: **procedure** FindPF(pts)
2: $result \leftarrow$ new array
3: Sort pts first ascending based on β_D, and then descending on β_A
4: Add $pts[0]$ to $result$
5: **for** $i \leftarrow 1$ to $|pts| - 1$ **do**
6: $(d, a) \leftarrow pts[i]$
7: $(d_{\text{last}}, a_{\text{last}}) \leftarrow result[|result| - 1])$
8: **if** $d > d_{\text{last}}$ and $a > a_{\text{last}}$ **then**
9: Add (d, a) to $result$
10: **return** $result$

Node	$\times_{u \in ch(v)} \text{BU}(T, u, \beta)$	Op.	Value pair	`FindPF`
d_1				$(0,0), (5, \infty)$
d_2				$(0,0), (10, \infty)$
AND_1	$\big((0,0),(0,0)\big)$, $\big((0,0),(10,\infty)\big)$, $\big((5,\infty),(0,0)\big)$, $\big((5,\infty),(10,\infty)\big)$	$\big(\sum, \min\big)$	$(0,0)$, $(10,0)$, $(5,0)$, $(15,\infty)$	$(0,0), (15,\infty)$
a_1				$(0,5)$
INH_1	$\big((0,0),(0,5)\big)$, $\big((15,\infty),(0,5)\big)$	$\big(\sum, \min\big)$	$(0,0)$, $(15,5)$	$(0,0), (15,5)$
a_2				$(0,10)$
INH_2	$\big((0,0),(0,10)\big)$, $\big((15,5),(0,10)\big)$	$\big(\sum, \sum\big)$	$(0,10)$, $(15,15)$	$(0,10), (15,15)$
a_3				$(0,15)$
OR_1	$\big((0,10),(0,15)\big)$, $\big((15,15),(0,15)\big)$	$\big(\sum, \min\big)$	$(0,10)$, $(15,15)$	$(0,10), (15,15)$

Table 4.2: Step-by-step walkthrough of the BU algorithm applied on Ex. 1.

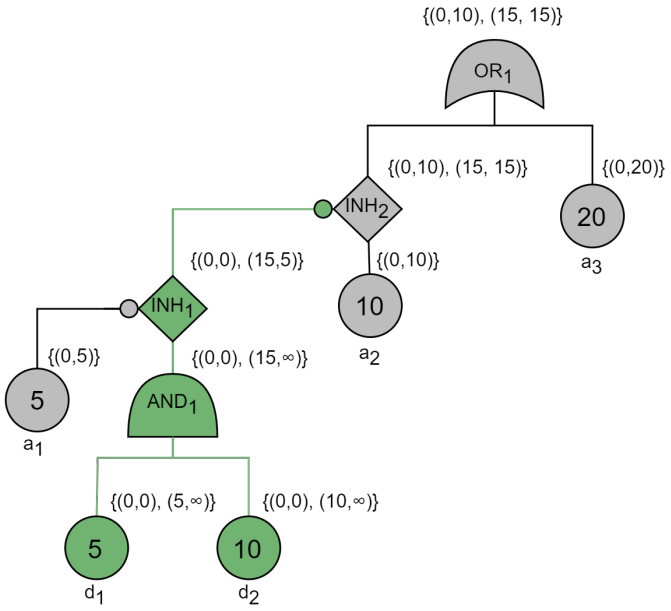


FIGURE 4.1: Graphical representation on how the BU algorithm propagates the Pareto fronts on the ADT from Ex. 1

.

As an alternative to the textual notation from Table 4.2, a graphical representation is created to illustrate how the values propagate in the tree towards the root node by finishing Ex. 1. This is depicted in Figure 4.1. In addition to the previously found $(15, 15)$ point, another Pareto point, $(0, 10)$, is observed, where the defender does nothing, and the minimum cost of a successful attack is achieved by activating a_2.

4.2 Complexity

Although Alg. 1 is linear in the size of the tree, the operations performed at each node are not linear because the Pareto fronts can grow exponentially due to the Cartesian product $\times$.

Example 3. Consider the ADT in Figure 4.2, where $R_T = OR(INH_1, INH_2, INH_3)$, and for each $w = INH_i$, $\beta_D(w_a) = \beta_A(w_c) = 2^i$. We strategically choose the values in β_D and β_A such that the sum between the defense costs is unique (and similarly for the attack costs). At each INH_i, we get $\mathrm{BU}(T, INH_i, \beta) = \{(0, 0), (2^i, 2^i)\}$. Taking the Cartesian product between n such `INH` nodes:

$$P = \left\{\left((x_1, x_1)(x_2, x_2), (x_3, x_3)\right) \mid x_i \in \{0, 2^i\}\right\}$$

The operators applied at a defense OR gate are $(\sum, \sum)$. As each element in $\left\{\sum_{i=1}^{3} x_i \mid x_i \in \{0, 2^i\}\right\}$ is unique, all the elements in P are Pareto optimal, resulting in an exponential Pareto front: $|\mathrm{PF}_S(T)| = 2^3$. Generalizing this for n is straightforward by adding additional `INH` nodes. Thus, for every $|\mathcal{D}|$, there is a T such that $|\mathrm{PF}_S(T)| = 2^{|\mathcal{D}|}$.

4.3 Proof for the Bottom-up Algorithm

Proof for Alg. 1. For an attack-defense tree T, we want to prove that Alg. 1 constructs the Pareto front as described in Section 3.4, which is $\mathtt{BU}(R_T) = \mathrm{PF}_S(T)$. To achieve this, we

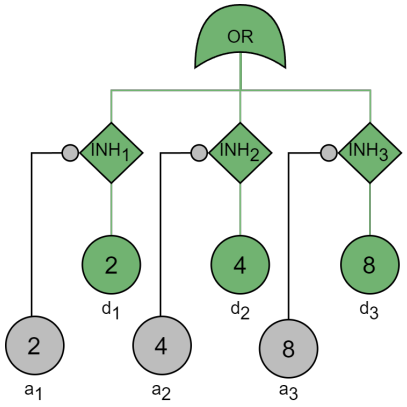


FIGURE 4.2: ADT depicting an exponential Pareto front size.

will use strong induction using the height of the nodes within the tree.

The height is a function $h\colon N \to \mathbb{N}$, where: $h(v) = \begin{cases} 1, & \text{if } v \in B \\ \max_{w \in ch(v)} h(w) + 1, & \text{if } v \notin B \end{cases}$

⋆ **Base case.** When the tree has a height of $h(v) = 1$, it consists of a single node $v = R_T$, which can be either a basic attack or defense step:

1. If $v \in \mathcal{A}$: There is a single strategy $\mathcal{S} = \{(0, v)\}$ with a value $\hat{\beta}\big((0, v)\big) = \big(0, \beta_A(v)\big)$. Therefore, $\mathrm{PF}_S(T) = \{(0, \beta_A(v))\}$ which is exactly what `BU` returns on line 4.

2. If $v \in \mathcal{D}$: The defender has two options: do nothing or activate v. If v is activated, then no attack is possible at this basic defense node, making the value of a "successful" attack ∞. This results in the Pareto front $\{(0, 0), (\beta_D(v), \infty)\}$, which is also the output of `BU` on line 6.

⋆ **Inductive Hypothesis.** Consider an ADT T with root $v = R_T$, having a height $h(v) = k$. Let T_w be the sub-tree rooted at a node $w \in N$. We assume that the property $\texttt{BU}(w) = \mathrm{PF}_S(T_w)$ holds for all children $w \in ch(v)$ at height $h(w) = k - 1$, and recursively for all descendants of these children.

⋆ **Induction step.** Using the inductive hypothesis, we want to prove that $\mathrm{BU}(v) = \mathrm{PF}_S(T_v)$ holds for v itself.

In the following, the proofs for Lemma 1 and 2 are postponed until the end of the section. The proof of Lemma 3 is straightforward and thus omitted. Likewise, Lemma 4 and 5 are derived from the structure of the `BU` and `FindPF` algorithms.

Lemma 1. $\hat{\beta}\big(\mathcal{S}(T_v)\big) = \left\{ \bigtriangledown_{i=1}^{n} \hat{\beta}(s_i) \mid \vec{s} \in \times_{i=1}^{n} \mathcal{S}(T_{v_i}) \right\}$

For a node $v \in N$ and a tree T_v, the set of strategies $\mathcal{S}(T_v)$ is constructed by considering all possible combinations of strategies from its child nodes v_i. Let $\bigtriangledown$ denote the suitable pairs of operators that should be applied for v (e.g. in the minimum cost domain, when $\gamma(v) = \texttt{OR}$ and $\tau(v) = A$ then $\bigtriangledown = (\sum, \min)$). The set of value pairs for strategies of T_v is determined by applying $\bigtriangledown$ to each tuple of strategies.

We start the induction step with this Lemma's equation:

$$\hat{\beta}\big(\mathcal{S}(T_v)\big) = \left\{ \bigtriangledown_{i=1}^{n} \hat{\beta}(s_i) \mid \vec{s} \in \bigtimes_{i=1}^{n} \mathcal{S}(T_{v_i}) \right\} \tag{4.1}$$

Lemma 2. $\left\{\bigtriangledown_{i=1}^{n}\hat{\beta}(s_i) \mid \vec{s} \in \times_{i=1}^{n}\mathcal{S}(T_{v_i})\right\} = \left\{\bigtriangledown_{i=1}^{n} g_i \mid \vec{g} \in \times_{i=1}^{n}\hat{\beta}\big(\mathcal{S}(T_{v_i})\big)\right\}$

We rewrite the right side of Equation 4.1 by moving the $\hat{\beta}$ to the predicate. In this context, $\vec{g}$ represents a vector of value pairs, on which the operations from $\bigtriangledown$ are applied:

$$\hat{\beta}\big(\mathcal{S}(T_v)\big) = \left\{\bigtriangledown_{i=1}^{n} g_i \mid \vec{g} \in \bigtimes_{i=1}^{n}\hat{\beta}\big(\mathcal{S}(T_{v_i})\big)\right\} \tag{4.2}$$

Apply Lemma 2 to replace the right-hand side of Equation 4.2. Additionally, we simplify the notation by applying $\bigtriangledown$ directly to the value pairs resulted from $\hat{\beta}$:

$$\hat{\beta}\big(\mathcal{S}(T_v)\big) = \bigtriangledown \bigtimes_{i=1}^{n}\hat{\beta}\left(\mathcal{S}(T_{v_i})\right) \tag{4.3}$$

Remove the dominated value pairs from both sides of the equation:

$$\underline{\min}_{\sqsubseteq}\hat{\beta}\big(\mathcal{S}(T_v)\big) = \underline{\min}_{\sqsubseteq}\bigtriangledown \bigtimes_{i=1}^{n}\hat{\beta}\left(\mathcal{S}(T_{v_i})\right) \tag{4.4}$$

Lemma 3. The $\underline{\min}_{\sqsubseteq}$ operation is idempotent. For a set of points X in a two-dimensional space, $\underline{\min}_{\sqsubseteq}(\underline{\min}_{\sqsubseteq}(X)) = \underline{\min}_{\sqsubseteq}(X)$.

We rewrite the left-hand side of the equation with the equivalent $\mathrm{PF}_S(T_v)$ notation. On the right-hand side, we apply Lemma 3 which permits us to take $\underline{\min}_{\sqsubseteq}$ of $\hat{\beta}\big(\mathcal{S}(T_{v_i})\big)$ without changing the results.

$$\mathrm{PF}_S(T_v) = \underline{\min}_{\sqsubseteq}\bigtriangledown \bigtimes_{i=1}^{n}\underline{\min}_{\sqsubseteq}\hat{\beta}\big(\mathcal{S}(T_{v_i})\big) \tag{4.5}$$

Now apply the $\mathrm{PF}_S(T_v)$ notation on the right side as well:

$$\mathrm{PF}_S(T_v) = \underline{\min}_{\sqsubseteq}\bigtriangledown \bigtimes_{i=1}^{n}\mathrm{PF}_S(T_{v_i}) \tag{4.6}$$

Use the Induction Hypothesis for each child v_i:

$$\mathrm{PF}_S(T_v) = \underline{\min}_{\sqsubseteq}\bigtriangledown \bigtimes_{i=1}^{n}\mathrm{BU}(v_i) \tag{4.7}$$

Lemma 4. $\underline{\min}_{\sqsubseteq}X = \mathrm{FindPF}(X)$

The algorithm `FindPF` sorts the elements in X in ascending order based on the defense cost and in descending order based on the attack cost, following the logic of $\sqsubseteq$. Then, the dominated elements are removed, representing the $\underline{\min}$ operation. Apply Lemma 4 to replace $\underline{\min}_{\sqsubseteq}$ on the right-hand side of the equation:

$$\mathrm{PF}_S(T_v) = \mathrm{FindPF}\Big(\bigtriangledown \bigtimes_{i=1}^{n}\mathrm{BU}(v_i)\Big) \tag{4.8}$$

Lemma 5. $\mathrm{BU}(T_v) = \mathrm{PF}\left(\bigtriangledown \times_{i=1}^{n}\mathrm{BU}(v_i)\right)$

The operations described by $\bigtriangledown$ are the same operations described in Table 4.1. In line 10 of the algorithm, FindPF is called with the aggregated metric values resulting from applying $\bigtriangledown$.

$$\mathrm{PF}_S(T_v) = \mathrm{BU}(T_v) \tag{4.9}$$

□

Proof of Lemma 1. Let $v_i \in ch(v)$ be a child node of v and $\mathcal{S}(T_{v_i})$ be the set of strategies in the sub-tree T_{v_i}. To find $\mathcal{S}(T_{v_i})$, each possible combination of strategies from each child node needs to be considered because a strategy for v depends on how all the strategies from all its children are combined. This can be achieved by taking the Cartesian product, denoted by $\times$, across all children i:

$$\bigtimes_{i=1}^{n} \mathcal{S}(T_{v_i})$$

To determine $\hat{\beta}(\mathcal{S}(T_v))$, each combination of strategies from $\times_{i=1}^{n} \mathcal{S}(T_{v_i})$ needs to be aggregated to a single value pair for the node v. This can be achieved by choosing the appropriate $\bigcirc_D$ and $\bigcirc_A$ operators from Table 4.1, based on the specific properties of the node v (e.g., $\gamma(v)$ and $\tau(v)$). An argument for the appropriateness of the operators is given in Section 4.1. We denote the operator pair $(\bigcirc_D, \bigcirc_A)$ by $\bigtriangledown$, leading to

$$\hat{\beta}\big(\mathcal{S}(T_v)\big) = \left\{ \bigtriangledown_{i=1}^{n} \hat{\beta}(s_i) \mid \vec{s} \in \bigtimes_{i=1}^{n} \mathcal{S}(T_{v_i}) \right\}$$

□

Proof of Lemma 2. To show that these two sets are equal, we need to prove that for any $\vec{s} \in \times_{i=1}^{n} \mathcal{S}(T_{v_i})$, there exists a corresponding $\vec{g} \in \times_{i=1}^{n} \hat{\beta}\big(\mathcal{S}(T_{v_i})\big)$ such that $\bigtriangledown_{i=1}^{n} \hat{\beta}(s_i) = \bigtriangledown_{i=1}^{n} g_i$ and vice versa. Recall that $\vec{s}$ represents a tuple formed by combining the strategies of all the sub-trees T_{v_i}. Let $\vec{s} = (s_1, s_2, \ldots, s_n) \in \times_{i=1}^{n} \mathcal{S}(T_{v_i})$. Each $s_i = (\vec{\delta_i}, \vec{\alpha_i})$ can be seen as a pair of binary vectors that $\hat{\beta}_D$ and $\hat{\beta}_A$ can act upon. Define:

$$\vec{\delta'} = \begin{pmatrix} \vec{\delta_1} \\ \vec{\delta_2} \\ \vdots \\ \vec{\delta_n} \end{pmatrix} \quad \text{and} \quad \vec{\alpha'} = \begin{pmatrix} \vec{\alpha_1} \\ \vec{\alpha_2} \\ \vdots \\ \vec{\alpha_n} \end{pmatrix}$$

so that $\vec{s} = (\vec{\delta'}, \vec{\alpha'})$. Given $\vec{s}$, we apply the operators $\hat{\beta}_D$ and $\hat{\beta}_A$ to the components $\vec{\delta'}$ and $\vec{\alpha'}$, respectively: $\vec{g} = \big(\hat{\beta}_D(\vec{\delta'}), \hat{\beta}_A(\vec{\alpha'})\big)$. By construction, $\vec{g} \in \times_{i=1}^{n} \hat{\beta}(\mathcal{S}(T_{v_i}))$. Applying the operators $\bigtriangledown_{i=1}^{n}$, we get:

$$\bigtriangledown_{i=1}^{n} \hat{\beta}(s_i) = \bigtriangledown_{i=1}^{n} g_i.$$

Conversely, let $\vec{g} \in \times_{i=1}^{n} \hat{\beta}\big(\mathcal{S}(T_{v_i})\big)$. For each i, there exists $s_i = (\vec{\delta_i}, \vec{\alpha_i}) \in \mathcal{S}(T_{v_i})$ such that $g_i = \big(\hat{\beta}_D(\vec{\delta_i}), \hat{\beta}_A(\vec{\alpha_i})\big)$. Define $\vec{s} = (s_1, s_2, \ldots, s_n) \in \times_{i=1}^{n} \mathcal{S}(T_{v_i})$. Applying the operators $\bigtriangledown_{i=1}^{n}$, we get:

$$\bigtriangledown_{i=1}^{n} g_i = \bigtriangledown_{i=1}^{n} \hat{\beta}(s_i)$$

Hence, the two sets are equal:

$$\left\{ \bigtriangledown_{i=1}^{n} \hat{\beta}(s_i) \mid \vec{s} \in \bigtimes_{i=1}^{n} \mathcal{S}(T_{v_i}) \right\} = \left\{ \bigtriangledown_{i=1}^{n} g_i \mid \vec{g} \in \bigtimes_{i=1}^{n} \hat{\beta}\big(\mathcal{S}(T_{v_i})\big) \right\}$$

□

Chapter 5

DAG-structured ADTs

The classical BU algorithm does not work when the ADT has a DAG structure [16]. This limitation also holds for the adapted BU algorithm in Alg. 1. An intuitive explanation is that when a node has multiple parents, the Pareto Front computed at that node is propagated multiple times up the tree, causing that value to be counted several times.

To illustrate this, we apply Alg. 1 on the ADT from Figure 5.1 and annotate it using a graphical representation. The node d_1 serves as a counter-attack for both inhibition gates, giving this ADT a DAG structure. At each inhibition node, the operators $\left(\sum, \sum\right)$ are applied on the combinations of value pairs, resulting in $\texttt{BU}(T, INH_1, \beta) = \big\{(0,2),(1,\infty)\big\}$ and $\texttt{BU}(T, INH_2, \beta) = \big\{(0,1),(1,\infty)\big\}$.

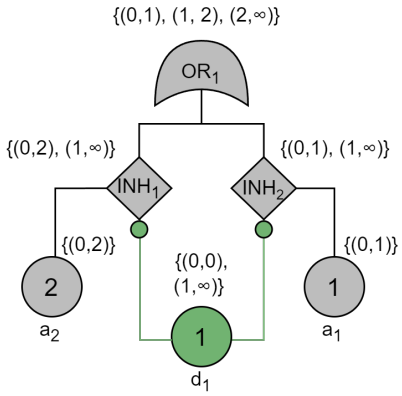


FIGURE 5.1: BU algorithm applied on a DAG, leading to incorrect results.

At the root OR_1 node, when $\text{PF}_S(INH_1)$ and $\text{PF}_S(INH_2)$ are combined, then d_1 will inevitably be counted twice.

$$\texttt{BU}(T, OR_1, \beta) = \big\{(0,1),(1,2),(2,\infty)\big\}$$

Furthermore, the value pair $(1,2)$ is produced, which is not valid: with a defense cost of at least 1, it is **incorrect** that the node OR_1 is not activated unless the attacker spends at least 2. In this scenario, it should be that *no attack can reach* OR_1 *when the defender spends at least 1.*

To find the correct Pareto Front for the tree in Figure 5.1, we need to find $\underline{\min}_{\sqsubseteq}\hat{\beta}(\mathcal{S})$. To achieve this, we create a table similar to Table 1.1 where we track all possible defenses, their costs, and minimum attacks. This is illustrated in Table 5.1, where we obtain $\hat{\beta}(\mathcal{S}) = \big\{(0,1),(1,\infty)\big\}$. All value pairs are already Pareto optimal, so:

$$PF_S(OR_1) = \{(0,1),(1,\infty)\}$$

Clearly, applying *Alg.* 1 yields incorrect results for DAGs since $\texttt{BU}(T, OR_1, \beta) \neq PF_S(OR_1)$.

Defense	**Defense cost**	**Min. size attacks**	**Min. attack cost**
$\varnothing$	0	$\{\{a_1\}, \{a_2\}\}$	1
$\{d_1\}$	1	$\varnothing$	∞

TABLE 5.1: Quantitative evaluation of the ADT in Figure 5.1

5.1 Naive approach

For ATs, it is known from Lopuhaä-Zwakenberg *et al.* [26] that computing a metric for a semiring attribute domain in a DAG-structured AT is generally NP-hard. The same holds for the ADTs discussed in this thesis, as they are an extension of ATs.

The procedure applied in Table 1.1 and Table 5.1 is clearly inefficient. To find $\mathcal{S}$, we go through each possible defense vector, find the minimum cost attack, and remove the dominated points according to $\sqsubseteq$.

The time complexity of this algorithm is primarily driven by the exponential growth of the number of combinations of attacks and defenses in the tree, which is $2^{|\mathcal{D}|+|\mathcal{A}|}$. At each step, computing $\hat{\beta}_D(\vec{\delta})$ takes $\mathcal{O}(|\mathcal{D}|)$, $\hat{\beta}_A(\vec{\alpha})$ takes $\mathcal{O}(|\mathcal{A}|)$, while $f_T(\vec{\delta}, \vec{\alpha}, R_T)$ (checking if the strategy is successful) is linear in the number of nodes in the tree: $\mathcal{O}(|N|)$. Bringing it all together, the Naive approach takes $\mathcal{O}(2^{|\mathcal{D}|+|\mathcal{A}|} \cdot (|\mathcal{D}| + |\mathcal{A}| + |N|))$ which is clearly inefficient for large-scale trees.

Even so, formally defining this approach is still beneficial: it provides a practical reference point for what $\underline{\min}_{\sqsubseteq} \hat{\beta}(\mathcal{S})$ outputs for DAGs, while serving as a stepping stone for the next algorithms to improve on. Alg. 2 is rather straight forward. Lines 4-11 compute $\rho(\vec{\delta})$ for each $\vec{\delta}$ by going through all the possible attacks and finding the one with the minimum cost. In the end, the value pairs are reduced to the Pareto front using `FindPF` (Lemma 4).

Algorithm 2 Naive algorithm for DAGs

Input:
T: attack-defense tree
v: node $v \in N$
β: assignment of nodes $\in N$
Output: Pareto front of the sub-tree rooted at v.

```
1:  procedure NAIVE(T, v, β)
2:      result ← new array
3:      for δ⃗ ∈ 2^D do
4:          att_costs ← new array
5:          for α⃗ ∈ 2^A do
6:              if f_T(δ⃗, α⃗, R_T) = 1 then          ▷ if the attack is successful
7:                  Add β̂_A(α⃗) to att_costs
8:          if att_costs = ∅ then
9:              Add (β̂_D(δ⃗), ∞) to result
10:         else
11:             Add (β̂_D(δ⃗), min(att_costs)) to result
12:     return FindPF(result)
```

Chapter 6

Biobjective Integer Linear Programming

Integer Linear Programming (ILP) is another method that can be used for quantitative analysis [34]. This involves a mathematical optimization problem where the variables are constrained to be integers, and the objective function and constraints are linear. When dealing with multiple objectives to optimize, the approach is called Multi-Objective Integer Linear Programming (MOILP). Given that we have two objectives to optimize (the defender's and attacker's cost), we can name our approach Biobjective Integer Linear Programming (BILP).

Each node in the tree is modelled as a binary variable [29]. This approach aligns with the goal of computing the Pareto front: $\vec{\delta}$ and $\vec{\alpha}$ indicate whether a defense, respectively, an attack, is enabled or disabled. To convert the BILP solutions into a Pareto front, the cost for each $(\vec{\delta}, \vec{\alpha})$ solution is computed, and then the dominated points are removed. This implies that by finding $\hat{\beta}(\mathcal{S})$ using BILP, the solutions can then be reduced to $\underline{\min}_{\sqsubseteq}\hat{\beta}(\mathcal{S})$ using Lemma 4.

To compute $\hat{\beta}(\mathcal{S})$, the ϵ-constraint method is used, where the constraints are varied on different ϵ values to explore all solutions [40]. Each ϵ corresponds to a different defense vector in $\mathcal{D}$. This requires solving $2^{|\mathcal{D}|}$ single-objective problems, each known to be NP-hard with a worst-case complexity of 2^n, where n is the number of variables in the model [41]. Given that the model has a variable for each node in the tree, the worst-case complexity of solving BILP is $\mathcal{O}(2^{|\mathcal{D}|n})$, where n is the number of nodes in the tree.

Although the theoretical worst-case complexity of solving BILP is higher than that of the Naive approach, practical performance is usually much better due to the advanced heuristics and optimization techniques developed over time [42]. We can leverage these optimizations by using state-of-the-art solvers such as CPLEX or Gurobi which solve MOILP problems by repeatedly solving single-objective integer linear programming problems [36].

Note 6.1. Formulating a BILP problem which directly computes $\underline{\min}_{\sqsubseteq}\hat{\beta}(\mathcal{S})$ is not possible. In general, a BILP problem for ADTs needs to solve problems of the form

$$\begin{aligned} \text{minimize} \quad & \sum_{d\in\mathcal{D}} \delta_d \beta_D(d) \\ \text{maximize} \quad & \min \sum_{a\in\mathcal{A}} \alpha_a \beta_A(a) \\ \text{subject to} \quad & \vec{x} \in \{0,1\}^N, \\ & A \cdot \vec{x} \leqslant \vec{b} \end{aligned}$$

where A is a constraint matrix containing the coefficients of the linear inequalities, and $\vec{b}$ is a vector that specifies the upper bounds for each constraint. Although some solvers do not support directly maximizing an objective, this can easily be modelled as minimizing the negative value of that objective.

Let $\vec{x}$ be the status vector of all nodes in the tree, the first objective be the defense cost, and the second objective the attack cost. Then, the defense cost is a linear function, but the attack cost is not. The ordering $(d_1, c_1) \sqsubseteq (d_2, c_2)$ requires $c_1 \geqslant c_2$ while $\rho(\vec{\delta})$ requires $c_1 \leqslant c_2$.

Example 4. The nonlinearity of the attacker's function comes from the fact that when T has a DAG structure, the same defense vector $\vec{\delta}$ might be evaluated across different paths, leading to multiple points with the same defense cost but different minimal costs. Consider Figure 6.1 where $\mathcal{D} = \{d_1, d_2\}$, $\beta_D(d_1) = 10$ and $\beta_D(d_2) = 10$. Two solutions are $(10, 2)$ and $(10, 3)$, where 2 is the minimum cost for $\vec{\delta} = \{d_1\}$ and 3 is the minimum cost for $\vec{\delta} = \{d_2\}$. In this case, $(10, 3)$ is dominates $(10, 2)$ since it enforces a higher attack cost.

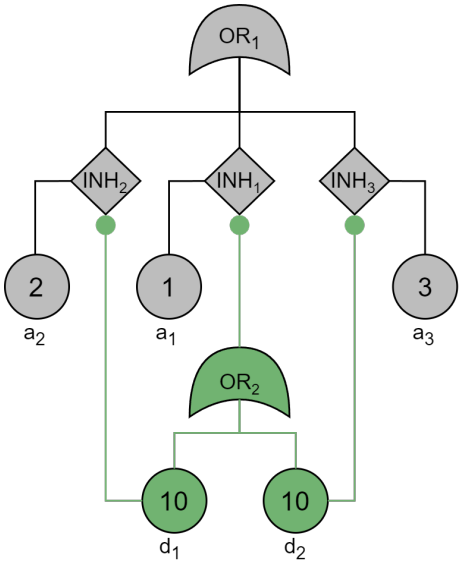


FIGURE 6.1: DAG representing non-linearity of attacker's cost function for BILP.

This forces us to find the max and min in two separate steps. Initially, the min operation is addressed by using a secondary objective function that minimizes $\hat{\beta}_A(\vec{\alpha})$. Then, the solutions with a minimum attack cost for each defense are passed to the `FindPF` function to discard the non-optimal elements.

Revisiting Figure 6.1, if we had not explored all the solutions using the ϵ-constraint method, then $(10, 3)$ would not have been a solution to begin with, as minimizing $\hat{\beta}_A(\vec{\alpha})$ would have excluded $(10, 3)$.

6.1 Variables

Since nodes in the tree are activated or disabled, there will be $|N|$ binary variables, denoted by $\vec{y} \in \mathbb{B}^N$. Using $\vec{y}$, the goal is to encode a vector pair $(\vec{\delta}, \vec{\alpha})$ such that for all $d \in \mathcal{D}$, $\delta_d = y_d$ and for all $a \in \mathcal{A}$, $\alpha_a = y_a$. For all the other variables $v \in N \backslash B$, linear constraints must be created such that y_v models $f_T(\vec{\delta}, \vec{\alpha}, v)$.

6.2 Constraints

For each gate $v \in N \backslash B$ with $\gamma(v) \in \{\texttt{AND}, \texttt{AND}, \texttt{INH}\}$, the following constraints are defined.

When v is an `AND` gate, we need to enforce $y_v = 1 \Leftrightarrow \forall w \in ch(v).\ y_w = 1$.

- $\forall w \in ch(v).\ y_v \leqslant y_w$: ensures that y_v is 1 only when all of its children are 1.
- $y_v \geqslant \big(\sum_{w \in ch(v)} y_w\big) - |ch(v)| + 1$: ensures that when all of its children are 1 then y_v is 1.

When v is an `OR` gate, we must enforce $y_v = 1 \Leftrightarrow \exists w \in ch(v).\ y_w = 1$.

- $\forall w \in ch(v).\ y_v \geqslant y_w$: ensures that when any of its children is 1, then y_v is 1.

- $y_v \leqslant \sum_{w \in ch(v)} y_w$: ensures that y_v is 1 when at least one of its children is 1.

When v is an `INH` gate, let v_a represent the attack child and v_c the counter-attack child. Then, we need to enforce $y_v = 1 \Leftrightarrow y_a = 1 \wedge y_c = 0$.

- $y_v \geqslant y_{v_a} - y_{v_c}$: ensures that y_v is 1 when the attack is 1 and the counter-attack is 0.
- $y_v \leqslant y_{v_a}$: ensures that when the attack is 0, y_v is also 0.
- $y_v \leqslant 1 - y_{v_c}$: ensures that when the counter-attack is 1, y_v must be 0.

Altogether, these constraints ensure that y_v has the behaviour specified by $f_T(\vec{\delta}, \vec{\alpha}, v)$ for all v.

Finally, we need one last constraint, called **goal**, to represent the attacker's intent: if $\tau(R_T) = \mathtt{A}$, then we need to make sure that R_T is reached: $y_{R_T} = 1$. Otherwise, R_T should not be reached: $y_{R_T} = 0$.

6.3 Objectives

The defender wants to minimize his own cost, which results in the first optimization function being to minimize $\sum_{d \in \vec{\delta}} \delta_d \beta_D(d)$.

As for the attacker, we revisit the definition of $\rho(\vec{\delta})$ from Section 3.5. Since the minimization constraint $f_T(\vec{\delta}, \vec{\alpha}, R_T) = 1$ (or $f_T(\vec{\delta}, \vec{\alpha}, R_T) = 0$ respectively) is represented by the **goal** constraint y_{R_T}, the attacker's cost is simply $\sum_{a \in \vec{\alpha}} \alpha_a \beta_A(a)$.

6.4 Model

Combining the previously mentioned variables, constraints, and objectives forms a BILP problem that finds the minimum attacker cost for a given $\vec{\delta}$. However, if there exists a $\vec{\delta}$ such that all attacks are prevented (as in Figure 5.1, where d_1 blocks all attacks), then BILP will have no feasible solution. In that case, the minimum cost is ∞.

For a defense vector $\vec{\delta}$, it is straightforward to prove that $\hat{\beta}_A\big(\rho(\vec{\delta})\big)$ can be found by solving the following BILP problem. If BILP has no feasible solution, then $\hat{\beta}_A\big(\rho(\vec{\delta})\big) = \infty$

$$
\begin{aligned}
\text{minimize}_{\vec{y} \in \mathbb{B}^N} \quad & \begin{pmatrix} \sum_{d \in \vec{\delta}} \delta_d \beta_D(d) \\ \sum_{a \in \vec{\alpha}} \alpha_a \beta_A(a) \end{pmatrix} \\
\text{subject to} \quad & \forall d \in \mathcal{D}. \\
& \qquad y_d = \delta_d, \\
& \forall v \in \{v' \in N \mid \gamma(v') = \mathtt{AND}\}. \\
& \qquad \forall w \in ch(v).\ y_v \leqslant y_w, \\
& \qquad y_v \geqslant \big(\sum_{w \in ch(v)} y_w\big) - |ch(v)| + 1, \\
& \forall v \in \{v' \in N \mid \gamma(v') = \mathtt{OR}\}. \\
& \qquad \forall w \in ch(v).\ y_v \geqslant y_w, \\
& \qquad y_v \leqslant \sum_{w \in ch(v)} y_w,
\end{aligned}
$$

$$
\begin{aligned}
&\forall v \in \{v' \in N \mid \gamma(v') = \texttt{INH}\}.\ y_{v_a}, y_{v_c} = ch(v) \text{ s.t. :} \\
&\quad y_v \geqslant y_{v_a} - y_{v_c}, \\
&\quad y_v \leqslant 1 - y_{v_c}, \\
&\quad y_v \leqslant y_{v_a}, \\
&y_{R_T} = 1 \text{ if } \tau(R_T) = \texttt{A};\ \text{otherwise } y_{R_T} = 0.
\end{aligned}
$$

For instance, the ADT and value assignment from Figure 6.1 produces the following problem for a fixed $\vec{\delta} = \{d_2\}$:

$$
\begin{aligned}
\text{minimize}_{\vec{y} \in \mathbb{B}^N} \quad & \begin{pmatrix} 10d_1 + 10d_2 \\ a_1 + 2a_2 + 3a_3 \end{pmatrix} \\
\text{subject to} \quad & \begin{aligned}
&y_{d_1} = 0, && y_{INH_1} \leqslant y_{a_1}, \\
&y_{d_2} = 1, && y_{INH_1} \geqslant 1 - y_{OR_2}, \\
&y_{OR_1} = 1, && y_{INH_1} \geqslant y_{a_1} - y_{OR_2}, \\
&y_{OR_1} \geqslant y_{INH_1}, && y_{INH_2} \leqslant y_{a_2}, \\
&y_{OR_1} \geqslant y_{INH_2}, && y_{INH_2} \geqslant 1 - y_{d_1}, \\
&y_{OR_1} \geqslant y_{INH_3}, && y_{INH_2} \geqslant y_{a_2} - y_{d_1}, \\
&y_{OR_1} \leqslant y_{INH_1} + y_{INH_2} + y_{INH_3}, && y_{INH_3} \leqslant y_{a_3}, \\
&y_{OR_2} \geqslant y_{d_1}, && y_{INH_3} \geqslant 1 - y_{d_2}, \\
&y_{OR_2} \geqslant y_{d_2}, && y_{INH_3} \geqslant y_{a_3} - y_{d_2}. \\
&y_{OR_2} \leqslant y_{d_1} + y_{d_2}, &&
\end{aligned}
\end{aligned}
$$

A few possible optimizations exist when implementing Alg. 3. For example, if a vector $\vec{\delta}$ results in an ∞ attack cost, then all the other defense vectors that extend $\vec{\delta}$ can be skipped, as they will also result in an ∞ attack cost while having a higher defense cost. However, these considerations are omitted here, as we are focusing on the general structure of the algorithm.

Algorithm 3 BILP

Input:
T: attack-defense tree
β: assignment of nodes $\in N$
Output: Pareto front of T.

```
1: procedure BILP(T, β)
2:     result ← new array
3:     model ← Initialize BILP problem, without any δ⃗ constraint
4:     for δ⃗ ∈ 2^D do
5:         Update model with δ⃗ constraint
6:         if model has a feasible solution then
7:             cost_sol ← retrieve model solution for the attacker
8:             Add (β̂_D(δ⃗), cost_sol) to result
9:         else
10:            Add (β̂_D(δ⃗), ∞) to result
11:    return FindPF(result)
```

Chapter 7

Binary Decision Diagrams

This section introduces three Binary Decision Diagrams (BDDs)-based algorithms to compute the Pareto front for ADTs with a DAG structure. Furthermore, we delve into the mathematical model behind BDDs and the constraints under which the algorithms operate. The performance differences between these algorithms are explored in Chapter 8.

BDDs offer a compact representation of Boolean functions. Since BDDS can have shared sub-trees, they can effectively model DAG structures. Generally, a BDD represents a Boolean function $f\colon \mathbb{B}^n \to \mathbb{B}$ over variables $Vars = \{x_i\}_{i=1}^n$ [26]. Specifically, for an ADT, $Vars$ contains the variables corresponding to defense steps, denoted by $\mathbb{D}$, and variables corresponding to attack steps, denoted by $\mathbb{A}$. Thus, $Vars = \mathbb{D} \cup \mathbb{A}$, where:

$$\mathbb{D} = \{d \in Vars \mid d \in \mathcal{D}\}$$
$$\mathbb{A} = \{a \in Vars \mid a \in \mathcal{A}\}$$

The structure function from Def. 4 allows an ADT to be modelled as a Boolean function, which can subsequently be represented by a BDD. We use Definition 5.8 of a BDD-encoded attack tree as outlined by Lopuhaä-Zwakenberg *et al.* [26]. This definition remains directly applicable for ADTs, as the underlying BDD model remains consistent although ADTs require a more complex Boolean function than ATs. For convenience, we will reiterate this definition in this section as well.

The terminal nodes in the BDD are binary values, specifically `False` and `True`, representing the Boolean function f output. For easier notation, 0_T is used for the `False` terminal node and 1_T for the `True` terminal node. Given that each variable in $Vars$ can appear multiple times in the BDD, it is essential to distinguish between a variable in f and a node within the tree. A nonterminal BDD node $w \in W$ represents a subfunction f_w of f as defined by its Shannon expansion. This means that w is associated with a variable $Lab(w) \in Vars$, a $Low(w)$ child representing f_w when $Lab(w)$ is set to 0, and a $High(w)$ child representing f_w when $Lab(w)$ is set to 1.

Definition 9. A BDD is a tuple $B = (W, Low, High, Lab)$ over a set $Vars$ where:

- The set of nodes W is partitioned into terminal nodes (W_t) and nonterminal nodes (W_n);
- $Low\colon W_n \to W$ maps each node to its *low* child;
- $High\colon W_n \to W$ maps each node to its *high* child;

- $Lab\colon W \to \{0_T, 1_T\} \cup Vars$ maps terminal nodes to Booleans and nonterminal nodes to variables:

$$Lab(w) \in \begin{cases} \{0_T, 1_T\} & \text{if } w \in W_t, \\ Vars & \text{if } w \in W_n. \end{cases}$$

Moreover, B satisfies the following constraints:

- (W, E) is a connected DAG, where

$$E = \left\{(w, w') \in W^2 \mid w' \in \{Low(w), High(w)\}\right\};$$

- B has a unique root, denoted R_B:

$$\exists! R_B \in W.\ \forall w \in W_n.\ R_B \notin \{Low(w), High(w)\}.$$

Definition 10. Given a BDD B and a Boolean vector of variables $\vec{x} \in \mathbb{B}^{Vars}$, a path in B corresponding to $\vec{x}$ is defined as a sequence of nodes $p = (w_0, w_1, \ldots w_m)$ where each nonterminal node $w_i \in W_n$ is associated with a Boolean variable x_i: $Lab(w_i) = x_i$. Additionally, p conforms to the following constraints:

- $w_0 \in W_n$ is the root node of the BDD.
- $w_m \in W_t$ is a terminal node.
- For each i where $0 \leqslant i < n$.
  - $w_{i+1} = Low(w_i)$ if $x_i = 0$
  - $w_{i+1} = High(w_i)$ if $x_i = 1$

7.1 Evaluating Path Costs

A path $p = (w_0, w_1, \ldots w_m)$ where $w_0 = R_B$ can correspond to multiple pairs of defense and attack vectors $(\vec{\delta}, \vec{\alpha})$:

1. For each w_i in p where $w_i \in \mathbb{D}$, the value of the Boolean variable $Lab(w_i)$ corresponds to the value of δ_{w_i}.
2. For each w_i in p where $w_i \in \mathbb{A}$, the value of the Boolean variable $Lab(w_i)$ corresponds to the value of α_{w_i}.
3. For each variable in $\mathbb{D} \cup \mathbb{A}$ that is not in p, that variable could either be activated or disabled since by construction of the BDD, the activation of this variable does not impact the output of the structure function.

As it is both the defender's and the attacker's goal to minimize their costs, it follows that if a variable w_i from $\mathbb{D} \cup \mathbb{A}$ is not in a path p, then w_i is disabled. As each variable in $\mathbb{D}$ and $\mathbb{A}$ is now mapped to a single Boolean, this implies that each path p corresponds to a unique strategy $(\vec{\delta}, \vec{\alpha})$. Consequently, the value pair of p can be found by taking $\hat{\beta}\big((\vec{\delta}, \vec{\alpha})\big)$.

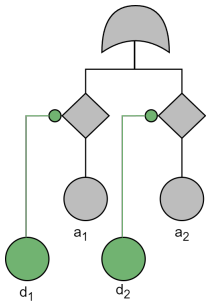


FIGURE 7.1: ADT for exemplifying variable ordering.

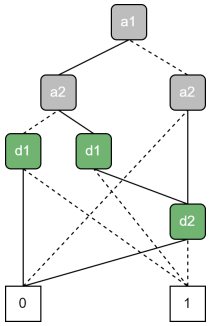


FIGURE 7.2: BDD with incorrect variable ordering.

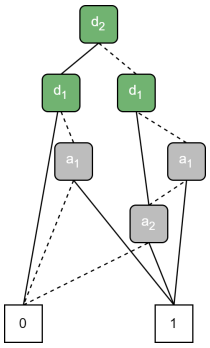


FIGURE 7.3: BDD with correct variable ordering.

7.2 Variable ordering

In this thesis, we work with Reduced Ordered BDDs (ROBDD). For a specification about what these entail, refer to Rudell [43]. Although a BDD can be uniquely reduced with any linear variable ordering $<$ defined on $Vars$, not all orderings can accurately represent ADTs with defensive and offensive attributes. These can be encoded as BDDs only for the orderings $<$ where all the defense variables precede all the attack variables.

Note 7.1. To model the assumption that the defender acts first, a linear variable ordering $<$ can be used for reducing a BDD-encoded ADT only if $\forall d \in \mathbb{D}. \forall a \in \mathbb{A}. d < a$. The linear orders used for the algorithms in this section must satisfy this property.

To understand the reasoning behind Note 7.1, consider the ADT in Figure 7.1 and its BDD representation in Figure 7.2, which uses the variable ordering $a_1 < a_2 < d_1 < d_2$. In Figure 7.2, grey nodes represent attack steps, while green nodes represent defense steps. The terminal nodes are marked with white. A dashed line from $w \rightarrow w'$ means that $Low(w) = w'$, while a solid line means $High(w) = w'$.

Note 7.2. When representing the path of a BDD in text, for clarity, we write w_i as $\neg w_i$ if $x_i = 0$, and w_i otherwise.

Although the BDD is reduced, the paths that end in 1_T do not always have minimum cost: the path $(a_1, a_2, \neg d_1, 1_T)$ leads to both a_1 and a_2 being activated, although only one of them is needed for reaching the root `OR` node. Furthermore, this ordering does not correctly model $\rho(\vec{\delta})$, as the attacker *does not know* the complete defense vector before choosing his actions.

As an alternative, the BDD in Figure 7.3 models Figure 7.1 using the variable ordering $d_2 < d_1 < a_1 < a_2$, which satisfies Note 7.1. In this case, $\rho(\vec{\delta})$ is correctly modelled, as at the first occurrence of an attack variable in the path, all defense variables have already been passed through.

Furthermore, the path $(d_2, d_1, 0_T)$ in this BDD has an interesting behaviour: it contains no attacks and leads to the terminal 0_T node. Any attacks that were previously in the BDD before it was reduced were removed, as their *Low* and *High* edges would lead to 0_T. This means that no $\vec{\alpha}$ can reach the root node when d_2 and d_1 are enabled.

Note 7.3. Given an ordering that satisfies Note 7.1 and a path $(w_1, \ldots, w_m)$ such that $w_m = 0_T$, and $\forall i < m \,.\, Lab(w_i) \in \mathbb{D}$, then $\rho(\vec{\delta}) = \infty$. In other words, if a path only contains defenses and ends in the terminal node 0_T, then the defenses on that path block all attacks.

7.3 Algorithms

The approach of encoding an ADT into a BDD is similar to that of encoding a fault tree into a BDD [44]. First, the propositional formula resulting from the structure function f_T is determined, and a variable ordering which satisfies Note 7.1 is identified. With these elements, the ROBDD can be constructed.

Similar to the BDD-based algorithms for attack trees, the attribute domain must be an absorbing semiring [26]. As specified in Section 3.5, we operate on the $(\mathbb{R}_{\geqslant 0}, \min, +)$ domain, which satisfies the proprieties of a semiring, and the absorption property (Section 3.2). When working with a BDD, there are multiple ways to compute $\underline{\min}_{\sqsubseteq} \hat{\beta}(\mathcal{S})$. The proofs for Theorems 7.1, 7.2, and 7.3 are outside the scope of this thesis.

7.3.1 BDD-PATHS

The first approach involves identifying all the paths in the BDD which lead to 1_T. For a node w, let $P(w)$ be the set of paths that start at node w and end at $\begin{cases} 1_T & \text{if } \tau(R_T) = \texttt{A} \\ 0_T & \text{if } \tau(R_T) = \texttt{D} \end{cases}$

Depending on the implementation, $P(w)$ can be found using a Depth-First-Search, Breadth-First-Search, Dynamic Programming or other approaches [45].

The BDD-PATHS algorithm is illustrated in Alg. 4. The first step is to create the ROBDD from the structure function f_T and a linear order "$<$". For each path in $P(R_{\text{bdd}})$, we compute the value pair (d, a) of the strategy that corresponds to $(\vec{\delta}, \vec{\alpha})$. In pf_dict, we keep track of all the previous paths and their defense costs. When pf_dict contains another path with the same defense cost d, there are two possibilities:

- If $\vec{\delta}$ and the defense vector corresponding to the other path are the same, then we model the behaviour of $\rho(\vec{\delta})$, which minimizes the attack cost.
- If the defense vectors differ, we encounter a situation similar to Figure 6.1 where we need to maximize the minimum attack cost.

Next, we search for all the paths leading to 0_T that only contain defenses (Note 7.3). If any are found, we add $\left(\hat{\beta}_D(\vec{\delta}), \infty\right)$ to *result*. Lastly, we remove all the dominated points using `FindPF`, resulting in the Pareto front.

Theorem 7.1. Given an ADT T, a cost assignment β, and a variable ordering "$<$" satisfying *Note* 7.1, Alg. 4 correctly computes the Pareto front: $\texttt{BDD}_{\texttt{PATHS}}(T, \beta, <) = \text{PF}_S(T)$.

Algorithm 4 BDD-PATHS

Input:
T: attack defense tree
β: assignment of nodes $\in N$
Output: Pareto Front of T.

procedure $\texttt{BDD}_{\texttt{PATHS}}(T, \beta, <)$
  $L_T \leftarrow$ Boolean expression given by f_T
  $bdd \leftarrow$ Create ROBDD from the expression L_T and order $<$
  $pf\_dict \leftarrow$ new dictionary
  for $p \in P(R_{\text{bdd}})$ **do**
    $(\vec{\delta}, \vec{\alpha}) \leftarrow p$ ▷ extract the vectors from the path
    $d \leftarrow \hat{\beta}_D(\vec{\delta})$
    $a \leftarrow \hat{\beta}_A(\vec{\alpha})$
    if d in pf_dict **then**
      $p' = pf\_dict[d]$
      $(\vec{\delta'}, \vec{\alpha'}) \leftarrow p'$
      $a' \leftarrow \hat{\beta}_A(\vec{\alpha'})$
      if $\vec{\delta} = \vec{\delta'}$ **then**
        if $a < a'$ **then** ▷ minimize attack if defense vectors are the same
          $pf\_dict[d] = p$
      else if $a > a'$ **then** ▷ maximize attack if defense vectors are different
        $pf\_dict[d] = p$
    else
      $pf\_dict[d] = p$
  $result \leftarrow$ Initialize with values of pf_dict.
  Add infinity points resulting from Note 7.3 to $result$
  return $\texttt{FindPF}(result)$

7.3.2 BDD-ALL-DEF

Another approach is to use the $\texttt{BDD}_{\texttt{DAG}}$ algorithm from Lopuhaä-Zwakenberg *et al.* [26] to find the minimum attacker cost. This requires running the algorithm $2^{|\mathcal{D}|}$ times by creating a BDD for the ADT obtained by fixing a defense vector $\vec{\delta}$ and performing a bottom-up computation on that BDD.

The BDD-ALL-DEF algorithm is illustrated in Alg. 5. To create a BDD from a fixed defense vector $\vec{\delta}$, let L_T be the Boolean expression given by the structure function f_T. Then, $L_T[\mathcal{D} \leftarrow \vec{\delta}]$ is the operation of substituting the defense variables in $\mathcal{D}$ with their Boolean values from $\vec{\delta}$.

Example 5. For an ADT T, let $L_T = (a_1 \wedge \neg d_1) \vee (a_2 \wedge \neg d_2)$ be the Boolean expression given by f_T. For $\vec{\delta} = \{d_2\}$, we replace d_1 with False and d_2 with True, resulting in:

$$L_T[d_1 \leftarrow \text{False}, d_2 \leftarrow \text{True}] = (a_1 \wedge \neg False) \vee (a_2 \wedge \neg True) = a_1$$

The BDD obtained from fixing $\vec{\delta}$ now embodies a regular AT. Now, it is possible to use the $\texttt{BDD}_{\texttt{DAG}}$ algorithm to compute the minimum cost. $\texttt{BDD}_{\texttt{DAG}}$ requires that the attribute domain is an idempotent and absorbing semiring, properties satisfied by the minimum cost (Section 3.5). The resulting value pair is added to $result$. Lastly, we remove all the dominated points using $\texttt{FindPF}$, yielding the Pareto front.

Theorem 7.2. Given an ADT T, a cost assignment β, and a variable ordering "$<$" satisfying *Note* 7.1, Alg. 5 correctly computes the Pareto front: $\texttt{BDD}_{\texttt{ALL-DEF}}(T, \beta, <) = \text{PF}_S(T)$.

Algorithm 5 BDD-ALL-DEF

Input:
T: attack defense tree
β: assignment of nodes $\in N$
Output: Pareto Front of T.

```
1: procedure BDD_ALL-DEF(T, β, <)
2:     result ← new array
3:     L_T ← Boolean expression given by f_T
4:     for δ⃗ ∈ 2^D do
5:         bdd ← Create ROBDD from L_T[D ← δ⃗], <
6:         att_cost ← BDD_DAG(bdd, R_T, β, (ℝ_⩾0, min, +))
7:         Add (β̂_D(δ⃗), att_cost) to result
8:     return FindPF(result)
```

7.3.3 BDD-BU

Lastly, we can adapt the $\texttt{BDD}_{\texttt{DAG}}$ algorithm from Lopuhaä-Zwakenberg *et al.* [26] to propagate the defender's costs together with the attacker's cost in a bottom-up manner. Similar to *Alg.* 1, we will eliminate the dominated points at each node $w \in W$.

The BDD-BU algorithm is illustrated Alg. 6. To focus on the recursivity of the algorithm, we omit the translation from the ADT to the BDD. This should be performed similarly to the other BDD algorithms by using a variable ordering "$<$" satisfying Note 7.1. Note that a BDD node w can be visited multiple times in this algorithm. Therefore, a significant optimization involves caching the computed Pareto front at w. However, this optimization is left out for a simpler algorithm.

The base cases of BDD-BU are defined by the terminal nodes in the BDD. At these nodes, the goal is to model the meaning of a value pair (Section 3.5). If the root of the ADT is an attacker ($t = \texttt{A}$ in the algorithm), then reaching 0_T means the attack was stopped by the defenses $(0, \infty)$. If 1_T is reached, then the attack succeeded $(0, 0)$.

When the root of the ADT is a defender, then the value pairs from above are swapped: reaching 0_T means the attack succeeded in stopping the defense: $(0, 0)$, while reaching 1_T implies that the defense stopped the attack: $(0, \infty)$.

Then, the Pareto fronts of the *low* and *high* children are computed in pf_low and pf_high respectively. Since taking a *high* child means that the variable $Lab(w)$ is True, we add $\beta_D\big(Lab(w)\big)$ to the defense cost (or $\beta_A\big(Lab(w)\big)$ to the attack cost) of each value pair in pf_high.

Finally, the non-optimal points in *result* are discarded in two different ways. When the variable of the BDD node is a defense step, this requires maximizing the attack cost through `FindPF`. However, when the variable is an attack step, we need to mimic the behaviour of $\rho(\vec{\delta})$ by finding all the attacks with minimum cost.

Theorem 7.3. Let b be a BDD that encodes an attack-defense tree T, reduced with a variable ordering "$<$" satisfying Note 7.1. Given a cost assignment β, Alg. 6 correctly computes the Pareto front: $\texttt{BDD}_{\texttt{BU}}(b, R_b, \beta, \tau(R_T)) = \text{PF}_S(T)$.

Algorithm 6 BDD-BU

Input:
bdd: Binary Decision Tree
w: node $\in W$
β: assignment of nodes $\in N$
t: value of $\tau(R_T)$
Output: Pareto Front of the ADT encoded by bdd.

```
1:  procedure BDD_BU(bdd, w, β, t)
2:      if (Lab(w) = 0_T ∧ t = A) ∨ (Lab(w) = 1_T ∧ t = D) then
3:          return {(0, ∞)}
4:      else if (Lab(w) = 0_T ∧ t = D) ∨ (Lab(w) = 1_T ∧ t = A) then
5:          return {(0, 0)}
6:      else                                                    ▷ w ∈ W_n
7:          pf_low ← BDD_BU(bdd, Low(w), β, t)
8:          pf_high ← BDD_BU(bdd, High(w), β, t)
9:          if Lab(w) ∈ 𝔻 then
10:             Add β_D(Lab(w)) to the defense of each element in pf_high
11:         else
12:             Add β_A(Lab(w)) to the attack of each element in pf_high
13:         result ← pf_low ∪ pf_high
14:         if Lab(w) ∈ 𝔻 then
15:             return FindPF(result)
16:         else                          ▷ Minimize the attack cost for each defense
17:             cost_dict ← new dictionary
18:             for (d, a) ∈ result do
19:                 if d not in cost_dict or a < cost_dict[d] then
20:                     cost_dict[d] ← a
21:             return items of cost_dict
```

7.4 Complexity

In the worst case, the size of a BDD is exponential in the number of $Vars$ [46]. As the linear ordering influences the BDD size, several heuristic approaches have been developed that find an optimal order [43, 47].

In our experiments, Rudell's sifting algorithm [43] generally resulted in the defenses preceding the attacks. This makes sense intuitively, as it avoids scenarios such as Figure 7.1, where paths contain unnecessary attack nodes that will be later defended against. However, it is not guaranteed that directly applying the existing heuristic will satisfy Note 7.1. Therefore, finding optimal variable ordering heuristics that satisfy Note 7.1 remains an open problem for future research.

Chapter 8

Experiments

In this section, we evaluate the performance of the algorithms presented in this thesis: `BU`, `Naive`, `BILP`, $\texttt{BDD}_{\texttt{PATHS}}$, $\texttt{BDD}_{\texttt{BU}}$, and $\texttt{BDD}_{\texttt{ALL-DEF}}$. Since this work introduces a novel model for representing attack-defense trees, existing literature and case-study examples cannot be directly used as they lack a defender's attribute domain.

One approach to address this issue is to annotate the existing ADTs found in the literature with cost values for the defender. We were able to find several ADTs with $25 \leqslant |N| \leqslant 50$ [48, 49] but only a limited number of ADTs with $50 \leqslant |N| \leqslant 100$ [12, 13, 50]. Finding ADTs with $|N| \geqslant 100$ can be challenging as they are not public for confidential reasons [51].

In practice, the size of attack trees can range from a few dozen to several hundred nodes [52]. Given the relatively small number and size of ADTs found in the literature, we do not consider this a sufficient testing suite to evaluate the algorithms. Consequently, it is necessary to generate ADTs synthetically. Two standard techniques for generating trees are combining literature trees into a single one [34] or generating random ones from scratch. Both approaches are used in our experiments, primarily focusing on the latter to cover a broader range of defense scenarios and create a more robust test suite.

A risk analysis algorithm that takes several days may be feasible for some applications. However, within the context of this thesis, given the hardware limitations and restricted time to conduct experiments, we limit our testing scope by not pursuing computations that take more than 10^4 seconds.

All experiments were performed on a machine with an Intel Core i5-12600K 3.7Ghz processor and 16GB of RAM. The algorithms are implemented in Python 3.12, and for BILP, we use the Gurobi solver [53] with the Python API. Although faster BDD run times could perhaps be achieved using a C implementation such as in Sylvan [54] or CUDD [55], we opted to maintain a consistent testing environment for all algorithms. The code and results are available on GitHub [1].

8.1 Random ADTs

This approach uses many randomly generated ADTs to statistically significantly compare the algorithms' performance. After setting a maximum number of children n, nodes with random proprieties (gate type, attack/defense type, number of children) are recursively generated until the tree has $|N| = n$ nodes. Refer to Appendix B for the specific algorithm used. This approach naturally creates tree- and DAG-structured ADTs.

[1] https://github.com/dvcopae/thesis_adtrees

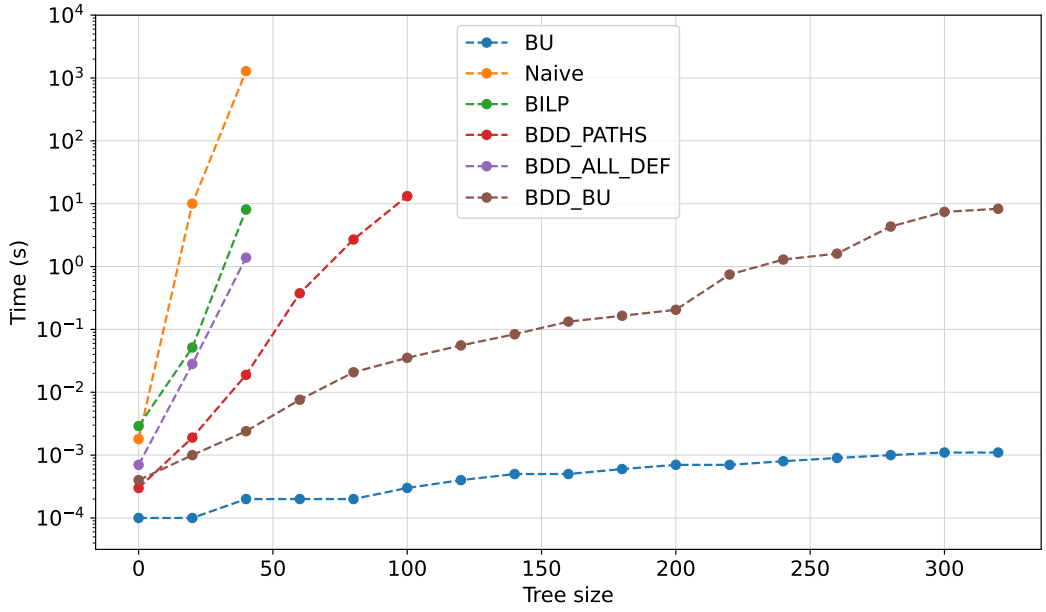


FIGURE 8.1: Summary of all the pairwise comparisons. The vertical axis represents each algorithm's median time in seconds for ADTs grouped by the number of nodes $|N|$ at intervals of size 20.

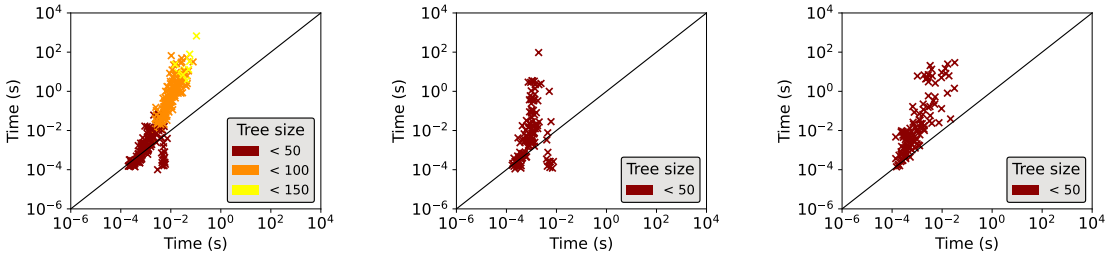
Figure 8.1 summarises the findings of the Random ADTs approach. In this plot, the algorithms' runtimes across all random graphs are aggregated by taking the median at each interval of $|N| = 20$. The number of ADTs in the interval, median and maximum runtime can be found in Appendix C. Since the run time of `Naive`, `BILP` and $\texttt{BDD}_{\texttt{ALL-DEF}}$ increases at an exponential rate, the values at the end of the interval will be drastically different from those at the beginning. Therefore, to better represent the central tendency of the interval, the median is used instead of the average.

The differences in the algorithms' performance are discussed in the following paragraphs, where we perform pairwise comparisons between all the algorithms. Note that Figure 8.1 uses the same ADTs as the pairwise comparisons, thus serving as a summary for them. The plots share several common characteristics:

- Each plot is labeled "x, y" where x is the algorithm on the horizontal axis, and y is the algorithm on the vertical axis.
- Both axes express run time in seconds.
- Each $\times$ on the plot represents a random ADT.
- If a $\times$ is placed on the lower side of the plot, it means algorithm y is faster; otherwise, x is faster.
- When one of the algorithms was `BU`, only tree-structured ADTs are generated.
- The size and number of random ADTs are adjusted to ensure no ADT takes more than 10^4 seconds.

All BDD-based algorithms are compared in Figure 8.2. For some trees with fewer than 50 nodes $\texttt{BDD}_{\texttt{BU}}$ appears to be equal to or even slightly slower than $\texttt{BDD}_{\texttt{PATHS}}$ and $\texttt{BDD}_{\texttt{ALL-DEF}}$.

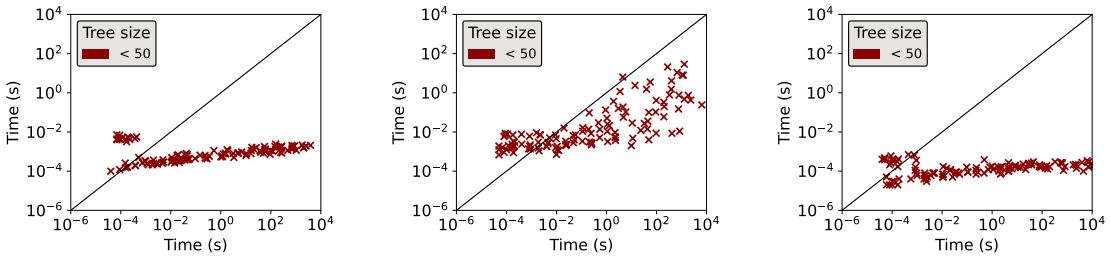
However, this difference is in the order of 10^{-2} seconds, which is unlikely to impact real-world scenarios significantly. Since $\texttt{BDD}_{\texttt{BU}}$ scales better as the number of nodes increases, $\texttt{BDD}_{\texttt{PATHS}}$ and $\texttt{BDD}_{\texttt{ALL-DEF}}$ will be omitted from future pairwise comparisons, as any algorithm faster than $\texttt{BDD}_{\texttt{BU}}$ will also be faster than these two.



(a) $\texttt{BDD}_{\texttt{BU}}$, $\texttt{BDD}_{\texttt{PATHS}}$ (b) $\texttt{BDD}_{\texttt{BU}}$, $\texttt{BDD}_{\texttt{ALL-DEF}}$ (c) $\texttt{BDD}_{\texttt{PATHS}}$, $\texttt{BDD}_{\texttt{ALL-DEF}}$

FIGURE 8.2: $\texttt{BDD}_{\texttt{BU}}$, $\texttt{BDD}_{\texttt{PATHS}}$, $\texttt{BDD}_{\texttt{ALL-DEF}}$ pairwise time comparison to determine the fastest BDD-based algorithm. Plot (a) contains 300 random ADTs with $|N| < 110$ while plot (b) and (c) 120 ADTs with $|N| < 45$.

For most of the small-sized trees with fewer than 50 nodes, the `Naive` algorithm surprisingly outperforms $\texttt{BDD}_{\texttt{BU}}$ and `BILP`. We hypothesize this is because, with a very small number of nodes, the time required to construct the BILP/BDD model makes up a significant proportion of the total run time. However, as the number of nodes increases, the `Naive` algorithm approaches a runtime of 10^4 seconds, even for some trees with less than 50 nodes. Clearly, the `Naive` algorithm has the slowest performance among the considered methods.

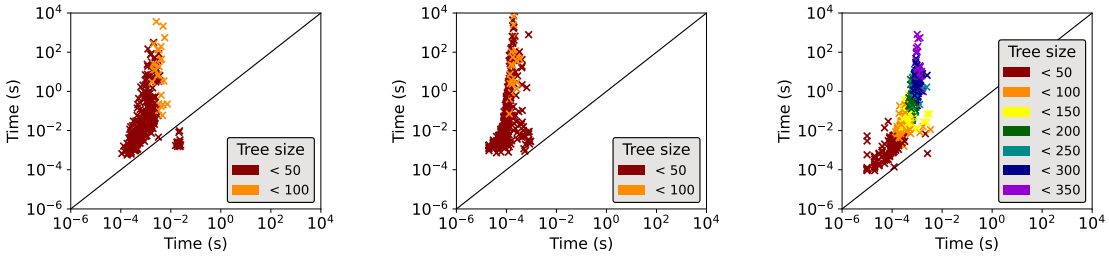


(a) `Naive`, $\texttt{BDD}_{\texttt{BU}}$ (b) `Naive`, `BILP` (c) `Naive`, `BU`

FIGURE 8.3: Pairwise time comparison between `Naive` and the other algorithms. All the plots in this figure are based on 120 random ADTs with $|N| < 45$.

In Figure 8.4, `BILP` is slower than both $\texttt{BDD}_{\texttt{BU}}$ and `BU`. While `BILP` allows for cost analysis of trees with 50 to 100 nodes within 10^4 seconds, it exceeds this time for larger trees. As both `BU` and $\texttt{BDD}_{\texttt{BU}}$ are quite fast, we extended our analysis for larger trees with up to 325 nodes. Remarkably, while the performance gap between the two remains consistent for trees with fewer than 100 nodes, this drastically changes as trees gain more nodes. For trees ranging from 300 to 325 nodes $\texttt{BDD}_{\texttt{BU}}$ requires approximately 10^3 seconds, whereas `BU` roughly 10^{-3} seconds.

In summary, the Random ADTs approach indicates that `BU` computes the Pareto Front the fastest for tree-structured ADTs, while $\texttt{BDD}_{\texttt{BU}}$ is the most efficient for DAGs.



(a) $\texttt{BDD}_{\texttt{BU}}$, BILP (b) BU, BILP (c) BU, $\texttt{BDD}_{\texttt{BU}}$

FIGURE 8.4: BU, BILP, $\texttt{BDD}_{\texttt{BU}}$ pairwise time comparison. Plot (a) and (b) are based on 300 random ADTs with $|N| < 60$ while plot (c) uses 600 ADTs with $|N| < 325$

8.2 Cloned ADTs

One of the drawbacks of the previous approach is that the difference in performance between the algorithms might stem from different tree structures. This approach runs all six algorithms on similarly structured trees of increasing size to verify whether this is the case. Given that BU only works for tree-structured ADTs, all trees must have a tree structure.

The following steps are followed to maintain a similar structure across subsequent trees. Starting with an ADT $T = (N, E)$, one of its attack leaves (BAS) is randomly replaced with another ADT T' that has the same acyclic graph structure but distinct instances for the nodes (refer to subsection 2.1.2 for the meaning behind the "instance of a node"). This substitution process is performed multiple times to generate sequences of trees that increase linearly in terms of $|N|, |\mathcal{D}|$ and $|\mathcal{A}|$.

As Figure 4.1 has a tree structure and comes with various gate types and attack/defense nodes, it is a suitable starting point for the cloning algorithm. This ADT has 9 nodes, of which 2 are basic defense steps. Applying the method outlined above, we generate trees that increase linearly by 8 in the number of nodes and 2 in the number of defenses. The results are presented in Figure 8.5. To replicate the size of the ADTs found in the literature, we cloned the tree from Figure 4.1, creating ADTs ranging from 9 to 129 nodes. As there is still some randomness in the cost assignments and the leaf replacements, we do not expect the run times to be strictly increasing but rather to have an increasing trend line. For the precise runtime details of the ADTs presented in this figure, refer to Appendix A.

To begin with, the BU algorithm computes the Pareto front the fastest, with a runtime of less than a millisecond, even for the largest tree containing 129 nodes. This algorithm scales the best with an increasing tree size, making it the best-performing choice when the ADT has a tree structure.

As anticipated, the Naive algorithm takes the longest time to run. For a tree with only 57 nodes, the algorithm requires over 3 hours, and thus, we intentionally stopped testing it for a higher number of nodes due to time constraints.

On a broader scale, the BILP and $\texttt{BDD}_{\texttt{ALL-DEF}}$ algorithms perform similarly. This is expected, as both of them are computing $\rho(\vec{\delta})$ for each possible $\vec{\delta}$. Unfortunately, they also surpass the 10^4 seconds threshold, albeit for an ADT of nearly double the size compared to the Naive approach. $\texttt{BDD}_{\texttt{ALL-DEF}}$ is slightly faster, requiring only 2.08 hours for an ADT with 97 nodes and 24 defenses, compared to 2.86 hours for BILP.

When comparing $\texttt{BDD}_{\texttt{PATHS}}$ and $\texttt{BDD}_{\texttt{BU}}$, the bottom-up approach has a significantly lower computation time. There are three factors that make $\texttt{BDD}_{\texttt{PATHS}}$ slower:

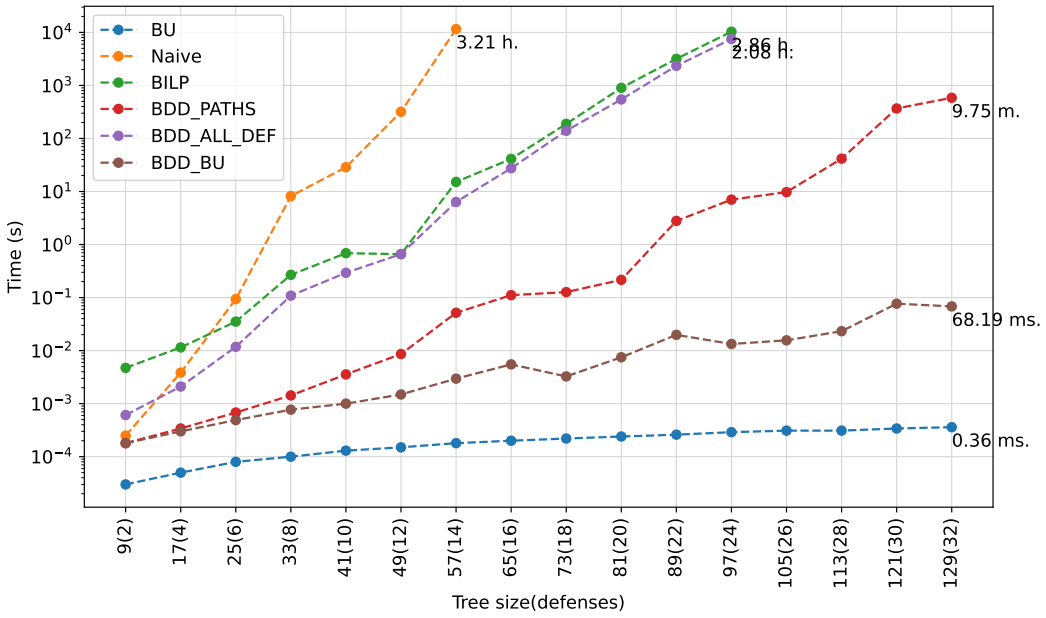


FIGURE 8.5: Algorithms runtime comparison on trees cloned from Figure 4.1. The labels on the horizontal axis follow the format "Tree size(defense count)" while the vertical axis is the runtime is seconds.

1. The implemented memoization for $\texttt{BDD}_{\texttt{BU}}$, which stores the Pareto fronts of intermediate nodes. This optimization is not feasible for $\texttt{BDD}_{\texttt{PATHS}}$ as there are no intermediate results to store for later reuse.

2. The addition of infinity points at the end using Note 7.3, which also requires analyzing all the failed paths.

3. In $\texttt{BDD}_{\texttt{BU}}$ all the non-optimal points are discarded at each BDD node w. However, in $\texttt{BDD}_{\texttt{PATHS}}$, the non-optimal points are discarded only for a specific path instead of all paths starting from a node w, leading to an exponential input for `FindPF` at the end.

The observations of the cloning approach align with those of the Random ADTs approach. `BU` is the fastest algorithm, capable of analyzing a tree-structured ADT with $|N| = 129$ under one millisecond. As for algorithms capable of analyzing DAGs, $\texttt{BDD}_{\texttt{BU}}$ computes the Pareto front the fastest, taking less than a millisecond for an ADT with $|N| = 129$. In contrast, other algorithms take minutes or even hours for the same ADT.

Chapter 9

Conclusion & Discussion

In this thesis, we introduced a novel framework designed to support defender's attribute domains within attack-defense trees. The primary highlights of this framework include a formal syntax and semantic model for representing ADT with attacker and defender attribute domains and modelling the Pareto front between these attribute domains.

Starting with a naive enumerative approach, we delved into various techniques, including bottom-up analysis, integer linear programming, and Binary Decision Diagram methods, for computing the Pareto front of the defender's and attacker's cost. We evaluated the performance of these approaches on a test suite of randomly generated ADTs with sizes of up to 325 nodes. We observed significant variations in the speed of the algorithms. However, most algorithms proved helpful under different ADT properties.

For tree-structured ADTs, the `BU` algorithm performs by far the best, with an average processing time of less than 1 second, even for trees with several hundred nodes. On the other hand, for ADTs with a DAG-structure absorbing semirings attribute domains, then $\texttt{BDD}_{\texttt{BU}}$ was the next fastest, capable of analyzing trees up to 325 nodes under 30 minutes.

When the attribute domains do not meet these proprieties, from the approaches analyzed in this thesis, `BILP` remains the only option for DAGs. Since the performance of `BILP` depends on the number of defense steps, giving expected time bounds solely in relation to the tree size will not be accurate. In our experiments, varying the number of defense steps from 0 to 50% of the total tree size, we found that on average `BILP` does not complete within 10^4 seconds for trees with > 100 nodes. In extreme cases, `BILP` is significantly faster when the ADT contains only attacks and significantly slower when there is only a single basic attack step.

Although the high runtime of the `Naive` approach makes it impractical for industrial applications, its simplicity could still make it a suitable tool for educational purposes, mainly because for very small ADTs (with approximately $|N| < 20$ nodes) it outperforms `BILP` and $\texttt{BDD}_{\texttt{BU}}$.

9.1 Future work

The algorithms presented in this thesis primarily focus on the minimum cost attribute domain for both the attacker and defender. Therefore, an exciting avenue to pursue in future studies is to explore how well the presented concepts generalize to other attribute domains. Below, we outline our initial considerations of the potential applicability of these algorithms to different attribute domains.

All the statements made in the thesis remain valid for metrics based on the same semiring as minimum cost (e.g. min. time in parallel). We hypothesize that the `BU`

algorithm can still be applied for other metrics by substituting the $\sum$ and min operators with those corresponding to the target attribute domains. Furthermore, given that a bottom-up approach is effective for DAG-structured ATs when both the $\oplus$ and $\otimes$ operations of the attribute domain are idempotent [26], it would be interesting to check whether this still property extends to ADTs.

As all the BDD-based methods operate on Reduced Ordered BDDs, these techniques do not apply to non-absorbing attribute domains [26]. In a BDD, all paths leading to 1_T represent successful attacks, yet not all are necessarily optimal. In the context of minimum cost (and absorbing semirings in general), the optimal attacks constitute a subset of the successful attacks, namely, all the minimum-length attacks. However, when considering maximum damage $(\mathbb{R}_{\geqslant 0}, \max, +)$, a non-absorbing metric, attacks can be optimal even if they are non-minimal. Consequently, the optimal attacks cannot be accurately modelled with BDDs.

Some adjustments are needed for `BILP` to accommodate other metrics. For instance, in the maximum challenge metric, $(\mathbb{R}_{\geqslant 0}, \max, \max)$ replacing $\oplus$ with max is pretty straightforward (i.e., minimize the negative amount value of the objective function). However, substituting $\otimes$ with a non-linear operation such as max proves to be complex, as the objective function is not linear anymore. However, one can utilize auxiliary variables and constraints to represent $\max(x_1, x_2, \ldots, x_n)$ as follows: introduce an auxiliary variable z and add constraints to ensure that z is at least at large as each x_i; the objective function would then involve maximizing z.

Another intriguing subject for future studies is identifying optimal variable orderings for $\texttt{BDD}_{\texttt{BU}}$, where all defense steps precede attack steps. In our experiments, we noticed that using Rudell's sifting algorithm to reduce the BDD resulted in such variable orderings. Therefore, formally verifying whether a minimum size BDD-encoded ADT (obtained through Rudell's algorithm or other heuristics) conforms to the desired order between defenders and attacks could be a promising area for future research.

Lastly, one could investigate whether other techniques, such as Bayesian networks [56] and game-theory-based methods [38] can be adapted to analyze DAGs with defender's and attacker's attribute domains.

Appendices

Appendix A

Experiments appendix

Tree Size (Defenses)	BU	Naive	BILP	BDD-PATHS	BDD-ALL-DEF	BDD-BU
9(2)	0.00003	0.00025	0.00473	0.00018	0.00061	0.00018
17(4)	0.00005	0.00385	0.01149	0.00034	0.00210	0.00030
25(6)	0.00008	0.09358	0.03520	0.00068	0.01177	0.00049
33(9)	0.00010	8.18713	0.26824	0.00143	0.10872	0.00077
41(10)	0.00013	28.74272	0.68756	0.00356	0.29321	0.00100
49(11)	0.00015	318.31062	0.65856	0.00862	0.66012	0.00149
57(14)	0.00018	11573.80061	15.15551	0.05165	6.34288	0.00296
65(16)	0.00020		41.20634	0.11108	27.49014	0.00550
73(18)	0.00022		188.44514	0.12682	139.59304	0.00326
81(20)	0.00024		899.42525	0.21593	544.23795	0.00749
89(22)	0.00026		3169.23115	2.78935	2338.29016	0.01984
97(24)	0.00029		10301.89091	7.02696	7505.23592	0.01334
105(26)	0.00031			9.74501		0.01563
113(28)	0.00031			41.62705		0.02323
121(30)	0.00034			368.13453		0.07677
129(32)	0.00036			585.01345		0.06819

FIGURE A.1: Performance metrics representing the runtime in seconds for all algorithms as tree sizes increase.

Appendix B

Random ADT algorithm

Create random ADT

1: $s \leftarrow 0$ # number of nodes already in the ADT

Input:
r: number of nodes that are reserved to be added in the ADT
n: maximum size of ADT

Output: Random ADT with a total size $< n$

2: **procedure** RandomADT(r, n)
3: **if** $s + r > n$ **then return** # No space for more nodes
4: **if** create DAG **then return** A random existing node
5: $v \leftarrow$ new node
6: $s \leftarrow s + 1$
7: Assign a random gate type $\gamma(v)$ to v
8: **if** $\gamma(v) =$ `BS` **then**
9: $\beta(v) \leftarrow$ random integer
10: **else if** $\gamma(v) =$ `INH` **then**
11: *attack* $\leftarrow$ RandomADT($r + 1, n$)
12: Add attack edge from v to *attack*
13: *counter* $\leftarrow$ RandomADT(r, n)
14: Add counter edge from v to *counter*
15: **else** # AND / OR gate type
16: *children_no* $\leftarrow$ randomly pick number of children
17: **for** $i \leftarrow 0$ to *children_no* **do**
18: $w \leftarrow$ RandomADT($r + (children\_no - i - 1), n$)
19: Add edge from v to w
20: **return** v

Appendix C

Random Trees Results

The tables from this appendix, the "Max" and "Median" rows illustrate the runtime in seconds. Each column represents an interval $[N, N + 20]$, where the column name denotes the start of the interval.

|N|	**0**	**20**	**40**	**60**	**80**	**100**
Items	209	200	131	35	39	37
Max	0.0022	0.0026	0.0013	0.0034	0.0005	0.0024
Median	0.0001	0.0001	0.0002	0.0002	0.0002	0.0003
	120	**140**	**160**	**180**	**200**	**220**
Items	40	36	38	39	32	40
Max	0.003	0.0008	0.0009	0.0009	0.0013	0.0024
Median	0.0004	0.0005	0.0005	0.0006	0.0007	0.0007
	240	**260**	**280**	**300**	**320**	
Items	36	47	25	29	7	
Max	0.0012	0.0015	0.0025	0.0015	0.0018	
Median	0.0008	0.0009	0.001	0.0011	0.0011	

TABLE C.1: Performance metrics for the BU method.

|N|	**0**	**20**	**40**
Items	174	158	28
Max	0.1127	888.9688	6515.199
Median	0.0018	10.0294	1284.8105

TABLE C.2: Performance metrics for the Naive method.

|N|	**0**	**20**	**40**
Items	276	284	158
Max	0.0914	148.2991	11804.3268
Median	0.0029	0.0515	8.086

TABLE C.3: Performance metrics for the BILP method.

|N|	**0**	**20**	**40**	**60**	**80**	**100**
Items	117	112	71	57	54	9
Max	0.0012	0.0168	0.4919	22.4812	65.9565	671.2649
Median	0.0003	0.0019	0.0189	0.3757	2.6911	13.1905

TABLE C.4: Performance metrics for the BDD-PATHS method.

|N|	**0**	**20**	**40**
Items	112	109	19
Max	0.1664	24.7026	95.3972
Median	0.0007	0.0282	1.3857

TABLE C.5: Performance metrics for the BDD-ALL-DEF method.

|N|	**0**	**20**	**40**	**60**	**80**	**100**
Items	319	322	198	93	93	46
Max	0.0248	0.0295	0.0129	0.0317	0.213	0.138
Median	0.0004	0.001	0.0024	0.0076	0.0208	0.0353
	120	**140**	**160**	**180**	**200**	**220**
Items	40	36	38	39	32	40
Max	0.3801	0.742	0.4673	4.0578	2.6165	12.9625
Median	0.0556	0.0837	0.133	0.1647	0.2053	0.7464
	240	**260**	**280**	**300**	**320**	
Items	36	47	25	29	7	
Max	11.8757	47.5158	92.5837	549.1171	811.3014	
Median	1.2932	1.6048	4.33	7.4171	8.2894	

TABLE C.6: Performance metrics for the BDD-BU method.

Bibliography

[1] C. A. E. II, "Fault tree analysis—a history," in *Proceedings of the 17th International Systems Safety Conference.*, Seattle, Washington, 1999.

[2] B. Schneier, "Attack trees," *Dr. Dobb's Journal of Software Tools*, vol. 24, no. 12, pp. 21–29, 1999, Accessed: May 2024. [Online]. Available: https://www.schneier.com/academic/archives/1999/12/attack_trees.html.

[3] B. Fila and W. Wideł, "Exploiting attack–defense trees to find an optimal set of countermeasures," in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, 2020, pp. 395–410. DOI: 10.1109/CSF49147.2020.00035.

[4] A. Buldas, P. Laud, J. Priisalu, M. Saarepera, and J. Willemson, "Rational choice of security measures via multi-parameter attack trees," in *Critical Information Infrastructures Security*, J. Lopez, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 235–248. DOI: 10.1007/11962977_19.

[5] *Amenaza securitree*, https://www.amenaza.com/documents.php, Accessed: May 2024.

[6] *Isograph attacktree*, https://www.isograph.com/software/attacktree/, Accessed: May 2024.

[7] E. Tanu and J. Arreymbi, "An examination of the security implications of the supervisory control and data acquisition (scada) system in a mobile networked environment: An augmented vulnerability tree approach.," 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:107872011.

[8] N. E. S. C. O. Resource, "Analysis of selected electric sector high risk failure scenarios," 2015, Version 2.0. [Online]. Available: https://smartgrid.epri.com/NESCOR.aspx.

[9] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Foundations of attack–defense trees," in *Formal Aspects of Security and Trust*, P. Degano, S. Etalle, and J. Guttman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 80–95. DOI: 10.1007/978-3-642-19751-2_6.

[10] B. Hyder and M. Govindarasu, "A novel methodology for cybersecurity investment optimization in smart grids using attack-defense trees and game theory," in *2022 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, 2022, pp. 1–5. DOI: 10.1109/ISGT50606.2022.9817467.

[11] X. Ji, H. Yu, G. Fan, and W. Fu, "Attack-defense trees based cyber security analysis for cpss," in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2016, pp. 693–698. DOI: 10.1109/SNPD.2016.7515980.

[12] M. Fraile, M. Ford, O. Gadyatskaya, R. Kumar, M. Stoelinga, and R. Trujillo-Rasua, "Using attack-defense trees to analyze threats and countermeasures in an atm: A case study," in *The Practice of Enterprise Modeling*, J. Horkoff, M. A. Jeusfeld, and A. Persson, Eds., Cham: Springer International Publishing, 2016, pp. 326–334, ISBN: 978-3-319-48393-1.

[13] A. Bagnato, B. Kordy, P. H. Meland, and P. Schweitzer, "Attribute decoration of attack–defense trees," *International Journal of Secure Software Engineering (IJSSE)*, vol. 3, pp. 1–35, Jan. 2012.

[14] E. A. Board and S. Board, "Threat trees and matrices and threat instance risk analyzer (tira)," 2015. [Online]. Available: https://www.eac.gov/sites/default/files/eac_assets/1/28/Election_Operations_Assessment_Threat_Trees_and_Matrices_and_Threat_Instance_Risk_Analyzer_(TIRA).pdf.

[15] S. Mauw and M. Oostdijk, "Foundations of attack trees," in *Information Security and Cryptology - ICISC 2005*, D. H. Won and S. Kim, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 186–198. DOI: 10.1007/11734727_17.

[16] B. Kordy and W. Wideł, "On quantitative analysis of attack–defense trees with repeated labels," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds., Cham: Springer International Publishing, 2018, pp. 325–346. DOI: 10.1007/978-3-319-89722-6_14.

[17] B. Fila and W. Wideł, "Efficient attack-defense tree analysis using pareto attribute domains," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019, pp. 200–20 015. DOI: 10.1109/CSF.2019.00021.

[18] R. Yang, C. Kiekintveld, F. Ordóñez, M. Tambe, and R. John, "Improving resource allocation strategies against human adversaries in security games: An extended study," *Artificial Intelligence*, vol. 195, pp. 440–469, 2013, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2012.11.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437021200152X.

[19] C. Sunstein, "Worst-case scenarios," *Bibliovault OAI Repository, the University of Chicago Press*, Jan. 2007.

[20] D. F. Haasl, N. H. Roberts, W. E. Vesely, and F. F. Goldberg, "Fault tree handbook," Jan. 1981. [Online]. Available: https://www.osti.gov/biblio/5762464.

[21] F. Arnold, H. Hermanns, R. Pulungan, and M. Stoelinga, "Time-dependent analysis of attacks," in *Principles of Security and Trust*, M. Abadi and S. Kremer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 285–305, ISBN: 978-3-642-54792-8.

[22] S. Bistarelli, F. Fioravanti, and P. Peretti, "Defense trees for economic evaluation of security investments," in *First International Conference on Availability, Reliability and Security (ARES'06)*, 2006, 8 pp.–423. DOI: 10.1109/ARES.2006.46.

[23] A. Roy, D. S. Kim, and K. S. Trivedi, "Attack countermeasure trees (act): Towards unifying the constructs of attack and defense trees," *Security and Communication Networks*, vol. 5, no. 8, pp. 929–943, 2012. DOI: 10.1002/sec.299.

[24] J. Arias, C. E. Budde, W. Penczek, L. Petrucci, T. Sidoruk, and M. Stoelinga, "Hackers vs. security: Attack-defence trees as asynchronous multi-agent systems," in *Formal Methods and Software Engineering*, S.-W. Lin, Z. Hou, and B. Mahony, Eds., Cham: Springer International Publishing, 2020, pp. 3–19.

[25] A. Bossuat and B. Kordy, "Evil twins: Handling repetitions in attack–defense trees," in *Graphical Models for Security*, P. Liu, S. Mauw, and K. Stolen, Eds., Cham: Springer International Publishing, 2018, pp. 17–37, ISBN: 978-3-319-74860-3.

[26] M. Lopuhaä-Zwakenberg, C. E. Budde, and M. Stoelinga, "Efficient and generic algorithms for quantitative attack tree analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 5, pp. 4169–4187, Sep. 2023, ISSN: 2160-9209. DOI: 10.1109/tdsc.2022.3215752.

[27] R. Jhawar, B. Kordy, S. Mauw, S. Radomirović, and R. Trujillo-Rasua, "Attack trees with sequential conjunction," in *ICT Systems Security and Privacy Protection*, H. Federrath and D. Gollmann, Eds., Cham: Springer International Publishing, 2015, pp. 339–353, ISBN: 978-3-319-18467-8.

[28] R. Kumar, E. Ruijters, and M. Stoelinga, "Quantitative attack tree analysis via priced timed automata," in *Formal Modeling and Analysis of Timed Systems*, S. Sankaranarayanan and E. Vicario, Eds., Cham: Springer International Publishing, 2015, pp. 156–171, ISBN: 978-3-319-22975-1.

[29] M. Lopuhaä-Zwakenberg and M. Stoelinga, *Cost-damage analysis of attack trees*, 2023. arXiv: 2304.05812 [cs.CR].

[30] F. Arnold, H. Hermanns, R. Pulungan, and M. Stoelinga, "Time-dependent analysis of attacks," in *Principles of Security and Trust*, M. Abadi and S. Kremer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 285–305, ISBN: 978-3-642-54792-8.

[31] M. Lopuhaä-Zwakenberg, *Attack tree metrics are operad algebras*, 2024. arXiv: 2401.10008 [cs.CR].

[32] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Attack-defense trees," *Journal of Logic and Computation*, vol. 24, Feb. 2014. DOI: 10.1093/logcom/exs029.

[33] J. Legriel, C. Le Guernic, S. Cotton, and O. Maler, "Approximating the pareto front of multi-criteria optimization problems," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 69–83, ISBN: 978-3-642-12002-2.

[34] M. Lopuhaä-Zwakenberg and M. Stoelinga, "Attack time analysis in dynamic attack trees via integer linear programming," in *Software Engineering and Formal Methods*, C. Ferreira and T. A. C. Willemse, Eds., Cham: Springer Nature Switzerland, 2023, pp. 165–183, ISBN: 978-3-031-47115-5.

[35] In *Applied Integer Programming: Modeling and Solution*. John Wiley & Sons, Ltd, 2009, ISBN: 9781118166000. DOI: 10.1002/9781118166000.

[36] M. Özlen and M. Azizoğlu, "Multi-objective integer programming: A general approach for generating all non-dominated solutions," *European Journal of Operational Research*, vol. 199, no. 1, pp. 25–35, 2009, ISSN: 0377-2217. DOI: 10.1016/j.ejor.2008.10.023.

[37] Dalton, Mills, Colombi, and Raines, "Analyzing attack trees using generalized stochastic petri nets," in *2006 IEEE Information Assurance Workshop*, 2006, pp. 116–123. DOI: 10.1109/IAW.2006.1652085.

[38] R. Jhawar, K. Lounis, and S. Mauw, "A stochastic framework for quantitative analysis of attack-defense trees," in *Security and Trust Management*, G. Barthe, E. Markatos, and P. Samarati, Eds., Cham: Springer International Publishing, 2016, pp. 138–153, ISBN: 978-3-319-46598-2.

[39] Z. Aslanyan and F. Nielson, "Pareto efficient solutions of attack-defence trees," in *Principles of Security and Trust*, R. Focardi and A. Myers, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 95–114, ISBN: 978-3-662-46666-7.

[40] P. Halffmann, L. E. Schäfer, K. Dächert, K. Klamroth, and S. Ruzika, "Exact algorithms for multiobjective linear optimization problems with integer variables: A state of the art survey," *Journal of Multi-Criteria Decision Analysis*, vol. 29, no. 5-6, pp. 341–363, 2022. DOI: 10.1002/mcda.1780.

[41] D. Knop, M. Pilipczuk, and M. Wrochna, "Tight complexity lower bounds for integer linear programming with few constraints," *ACM Trans. Comput. Theory*, vol. 12, no. 3, Jun. 2020, ISSN: 1942-3454. DOI: 10.1145/3397484. [Online]. Available: https://doi.org/10.1145/3397484.

[42] R. Bixby, "Solving real-world linear programs: A decade and more of progress," *Operations Research*, vol. 50, Oct. 2001. DOI: 10.1287/opre.50.1.3.17780.

[43] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, 1993, pp. 42–47. DOI: 10.1109/ICCAD.1993.580029.

[44] A. Rauzy, "New algorithms for fault trees analysis," *Reliability Engineering & System Safety*, vol. 40, no. 3, pp. 203–211, 1993, ISSN: 0951-8320. DOI: https://doi.org/10.1016/0951-8320(93)90060-C.

[45] G. Charwat and S. Woltran, "Bdd-based dynamic programming on tree decompositions," Database and Artificial Intelligence Group, TU Wien, Tech. Rep. DBAI-TR-2016-95, 2016. [Online]. Available: https://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-95.pdf.

[46] Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986. DOI: 10.1109/TC.1986.1676819.

[47] M. Lê, J. Weidendorfer, and M. Walter, "A novel variable ordering heuristic for bdd-based k-terminal reliability," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 527–537. DOI: 10.1109/DSN.2014.55.

[48] K. S. Edge, G. C. Dalton, R. A. Raines, and R. F. Mills, "Using attack and protection trees to analyze threats and defenses to homeland security," in *MILCOM 2006 - 2006 IEEE Military Communications conference*, 2006, pp. 1–7. DOI: 10.1109/MILCOM.2006.302512.

[49] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer, "Dag-based attack and defense modeling: Don't miss the forest for the attack trees," *Computer Science Review*, vol. 13-14, pp. 1–38, 2014, ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2014.07.001.

[50] K. Edge, R. Raines, M. Grimaila, R. Baldwin, R. Bennington, and C. Reuter, "The use of attack and protection trees to analyze security for an online banking system," in *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007, 144b–144b. DOI: 10.1109/HICSS.2007.558.

[51] S. Paul, "Towards automating the construction & maintenance of attack trees: A feasibility study," *Electronic Proceedings in Theoretical Computer Science*, vol. 148, pp. 31–46, Apr. 2014, ISSN: 2075-2180. DOI: 10.4204/eptcs.148.3. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.148.3.

[52] R. Vigo, F. Nielson, and H. R. Nielson, "Automated generation of attack trees," in *2014 IEEE 27th Computer Security Foundations Symposium*, 2014, pp. 337–350. DOI: 10.1109/CSF.2014.31.

[53] Gurobi Optimization, LLC, *Gurobi optimizer reference manual*, 2024. [Online]. Available: http://www.gurobi.com.

[54] T. v. Dijk and J. Van de Pol, "Sylvan: Multi-core framework for decision diagrams," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2016, pp. 677–691. [Online]. Available: https://github.com/trolando/sylvan.

[55] F. Somenzi, *Cudd: Cu decision diagram package*, Release 3.0.0, 2015. [Online]. Available: https://github.com/ivmai/cudd.

[56] Z. Wang, Y. Lu, X. Li, and W. N. and, "Optimal network defense strategy selection based on markov bayesian game," *KSII Transactions on Internet and Information Systems*, vol. 13, no. 11, pp. 5631–5652, Nov. 2019. DOI: 10.3837/tiis.2019.11.020.