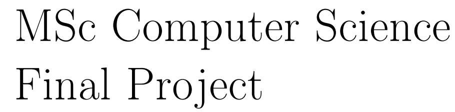

MSc Computer Science
Final Project

Transformation of Attack-Defense Trees into 2-player Game Automata

Kirill Fedorov

Supervisor: Milan Lopuhaä-Zwakenberg
Reza Soltani

Examiner: Mariëlle Stoelinga
Florian Hahn

August, 2025

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

UNIVERSITY OF TWENTE.

Contents

1 Introduction 1
1.1 Attack-Defense Trees . . . 2
1.2 Quantitative Analysis on Attack Trees . . . 2
1.3 Quantitative Analysis on Attack-Defense Trees . . . 3
1.4 Limitations of Existing ADT Analysis Approaches . . . 4
1.5 Applicability of PRISM . . . 5
1.6 Research Goal . . . 5

2 Background and related work 6
2.1 Attack-Defense Trees . . . 6
2.1.1 Quantitative analysis on ADTs . . . 10
2.1.2 Pareto Front . . . 13
2.1.3 Attack Trees vs Attack-Defense Trees . . . 13
2.2 Approaches for ADT analysis . . . 15
2.2.1 Existing ADT Analysis Tools . . . 15
2.2.2 Single metric computation . . . 17
2.2.3 Pareto fronts between multiple metrics on the same party . . . 18
2.2.4 Pareto fronts between attacker and defender . . . 20
2.3 Shortcomings in Existing Approaches . . . 23

3 Tool overview 24
3.1 Choice for model checkers . . . 24

4 ADTransformer: Tool Design and Implementation 26
4.1 Tool Architecture . . . 26
4.2 ADTool to PRISM: Transformation Workflow . . . 27
4.2.1 PRISM code structure . . . 28
4.2.2 Parser . . . 29
4.2.3 PRISM Model Generation . . . 29
4.3 Transformation results . . . 33
4.4 Queries . . . 35
4.5 Summary . . . 36

5 Pareto Front Analysis 37
5.1 Property-Guided Pareto Analysis . . . 37
5.2 How ADTransformer collects and processes the information . . . 38
5.2.1 On the Limitations of Alternative Properties . . . 39
5.2.2 Optimizations in Pareto Front Computation . . . 39
5.2.3 Case study . . . 40

5.3 Summary . . . 41

6 Experimental Evaluation **42**
6.1 Case study . . . 42
6.1.1 Queries . . . 42
6.1.2 Pareto front . . . 44
6.2 Test Environment . . . 46
6.3 Model Generation . . . 46
6.3.1 Test setup . . . 46
6.3.2 Model Generation Time . . . 47
6.3.3 Lines of Code Analysis . . . 47
6.4 Pareto Front computation . . . 48
6.4.1 Test setup . . . 48
6.4.2 Runtime computation . . . 48

7 Discussion & Conclusion **50**
7.1 Advantages of ADTransformer . . . 50
7.2 Limitations . . . 50
7.3 Future Work . . . 51
7.4 Summary . . . 51

8 Appendix A **53**

Abstract

Attack-Defense Trees (ADTs) are widely used for modeling dynamic interactions between a potential system attacker and the system's defender. They offer a structured and intuitive approach to quantitative analysis, enabling the identification of threats and the evaluation of defense strategies using various attributes. However, many existing approaches to ADT analysis suffer from a significant limitation—the restricted expressiveness of the models, which hinders in-depth analysis of ADTs. This creates a scientific gap in the field of cybersecurity and system analysis.

This thesis aims to address this gap by developing a novel generative tool that automates the creation of ADT models into PRISM code for formal verification. The tool will support both tree-structured and DAG-structured ADTs with cost metrics. Using this tool, it will be possible to conduct in-depth analysis of the trees, identify potential vulnerabilities, and model hypothetical scenarios for their prevention.

Keywords: attack-defense trees, PRISM, model checking

Chapter 1

Introduction

Protecting systems such as cyber-physical systems, critical infrastructures, smart power grids and etc. from potential threats has become a top priority for both individuals and organizations in the rapidly evolving field of cybersecurity. Modern systems are vulnerable to numerous threats due to their complexity and interconnectivity, necessitating a robust approach to cybersecurity management. Obviously, the need to close system vulnerabilities has driven the development of powerful tools and frameworks for model analysis, aimed at preventing potential attacks by proactively identifying and eliminating weaknesses.

Fault Trees (FTs), introduced back in the 1960ś [2], are a well-established formalism for analyzing the causes of system failures using hierarchical, tree-like structures. They allow analysts to reason about how basic component failures can propagate and lead to undesired system-level events. Building upon this foundational concept, Attack Trees (ATs), introduced by Schneier [36] adapt the same hierarchical modeling style for analyzing security threats. In ATs, nodes represent attacker goals and sub-goals, allowing the decomposition of complex attack scenarios into more manageable steps. This makes ATs particularly well-suited for systematically exploring potential vulnerabilities in a system and reasoning about attacker strategies. Moreover, their intuitive structure and analytical potential have led to a variety of extensions and applications, including integration with formal methods [37], game theory [17], and quantitative evaluation techniques [6]. In this thesis, we focus on the transformation of ATs into game-based models to support such analysis in a structured and scalable way.

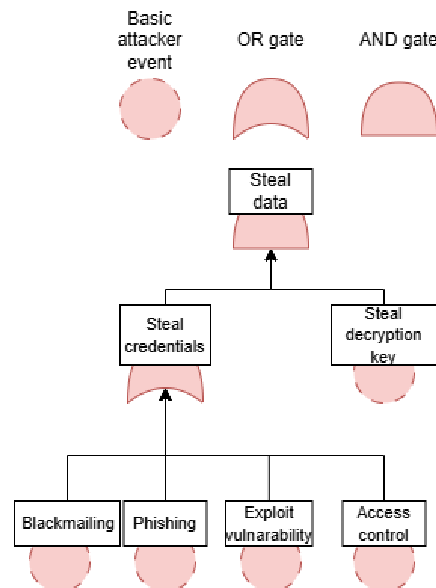

FIGURE 1.1: Attack tree depicting a scenario where the attacker aims to steal user data.

Fig. 1.1 illustrates an Attack Tree (AT) with a simple structure. This tree represents a logical depiction of a data theft scenario. To obtain the data, the attacker must both steal the decryption key and steal credentials, since Steal data is represented as an AND gate, meaning that all its sub-actions must be completed. At the same time, Steal credentials is modeled as an OR gate, combining actions such as blackmailing, phishing, vulnerability exploitation, and access control manipulation. Therefore, to steal the data, an attacker could, for instance, perform a phishing attack and

steal the decryption key, or exploit a vulnerability and steal the decryption key. In this way, ATs model a system and the potential attack scenarios symbolically and structurally.

1.1 Attack-Defense Trees

One of the major limitations of Attack Trees is that they do not account for defense mechanisms within the system, modeling only the behavior of the attacker. Kordy et al. introduced an extension to traditional ATs—Attack-Defense Trees (ADTs)—which addresses this limitation by integrating defense nodes [24]. These nodes act as countermeasures to the attacker nodes. ADTs have found wide application in the analysis and modeling of Cyber-Physical Systems [25], Information Systems [26], and Industrial Systems [20].

As stated in [10], The main advantage attack-defense trees have over attack trees is their ability to model defenses. Each basic attacker event has a binary activation value, where activated basic attacker events are part of an attack and deactivated basic attacker events are not. The defenses work by deactivating the attack nodes they are associated with, thereby disabling them.

Figure 1.2 presents an ADT. This system is an extended version of the system shown in Figure 1.1, with integrated defense mechanisms (Anti-phishing user training and Store Key Offline). The Anti-phishing user training node is a countermeasure to the Phishing Attack, and the Store Key Offline node is a countermeasure to the Steal Decryption Key attack. These defense nodes function in such a way that, if they are activated, the attacker—despite activating the corresponding attack nodes—will not be able to compromise the system through these specific attacks.

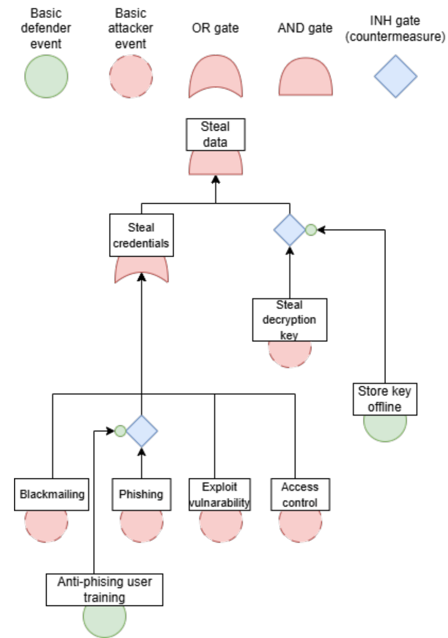

FIGURE 1.2: Attack-Defense tree where the attacker aims to steal user data, and the defender can prevent attack nodes from being propagated.

1.2 Quantitative Analysis on Attack Trees

The AT structure allows for quantitative analysis of attacker metrics [31]. Each basic attacker event is assigned a measurable value that describes the attack strategy, such as the minimum cost or the time required to perform the attack. The assigned values are included in the calculation when the basic attacker events are activated by the attacker.

Fig 1.3 is an extension of Fig 1.1 with the addition of costs to the basic attacker events. In this case, the attack cost is calculated using a bottom-up approach. The *"Steal credentials"* gate is an OR gate – this means that to activate this gate, it is sufficient for at least one child node to be fulfilled. Since the attacker aims to minimize the attack cost, the attacker will choose the scenario with the lowest cost, i.e., *"Phishing"* with a cost of 5.

The *"Steal data"* gate is an AND gate. This means that it requires all of its child nodes to be fulfilled – in this particular scenario, this includes *"Steal credentials"* and *"Steal*

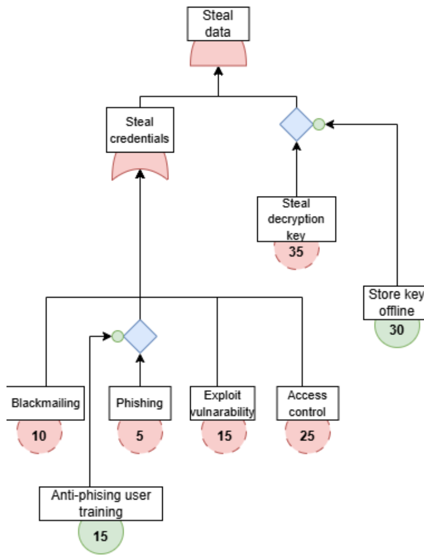

FIGURE 1.4: Attack-defense tree annotated with cost values where the attacker aims to steal user data, and the defender can prevent attack nodes from being propagated

decryption key". Since *"Steal decryption key"* has a cost of 35, the minimum cost to Steal data is the strategy with the set *{Phishing, steal decryption key}* with a total cost of 40.

1.3 Quantitative Analysis on Attack-Defense Trees

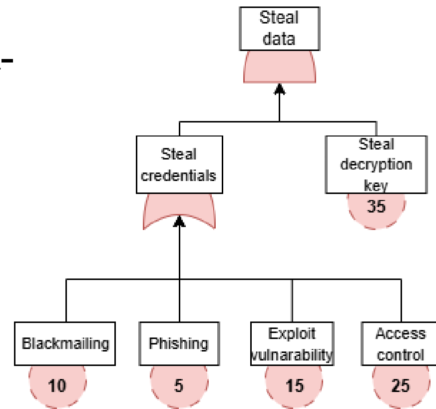

FIGURE 1.3: Attack tree annotated with cost values, depicting a scenario where the attacker aims to steal user data.

The ADT in Fig. 1.4 is an extension of the ADT from Fig. 1.2 with assigned costs for all basic events (attacker and defender). Like AT, ADT is subject to quantitative analysis. However, an important distinction is that while AT analysis typically identifies a single optimal attack strategy, ADT analysis involves evaluating multiple combinations of attacker and defender strategies. The minimum defense cost from the defender's perspective is also calculated. The minimum defense cost represents the least amount of resources the defender needs to allocate to ensure full protection of the system.

This approach involves evaluating all possible combinations of attacker strategies and selecting those defender strategies that can provide protection. All calculations are performed under the assumption that both the attacker and the defender have unlimited resources. The following table describes the combination of strategies:

From Table 1.1, it is evident that the defender strategies in rows 3 and 4 lead to the same result — the attacker has no possibility to bypass the defense, which results in an "infinite" attack cost. However, the strategy in row 3 requires fewer resources than the

Row	Defense	Defense Cost	minimal attacks	Min attack cost
1	$\emptyset$	0	{blackmailing, store decryption key}, {Phishing, store decryption key}, {Exploit valnurability, store decryption key}, {Access control, store decryption key}	40
2	{Anti-phishing use training}	15	{blackmailing, store decryption key}, {Exploit valnurability, store decryption key}, {Access control, store decryption key}	40
3	{Store key offline}	30	$\emptyset$	∞
4	{Store key offline, Anti-phishing use training}	45	$\emptyset$	∞

Table 1.1: Quantitative analysis of defense strategies

one in row 4 (30 versus 45). Since the defender, like the attacker, operates under the goal of minimizing the budget while maximizing the chances of protection, the strategy in row 3 is optimal.

1.4 Limitations of Existing ADT Analysis Approaches

Despite significant progress in the development of algorithms and tools for ADT analysis, current methods remain limited when it comes to partial analysis of trees. Most existing methods are designed to compute metrics or perform analysis on the entire tree, rather than conduct targeted analysis of subtrees or individual components. This is clearly observed in the literature, which describes bottom-up computation and metric evaluation over the entire tree, traversing from the leaves to its root [10, 4].

For example, the bottom-up algorithm, which is one of the most common and effective for quantitative analysis of ADTs, does not allow partial analysis of the tree [10, 4]. As a result, analysts cannot answer questions related to the impact of changes in the tree without fully modifying its structure. This is also observed in existing tools for ADT analysis, which do not support partial or modular analysis [38].

Despite the fact that many advanced algorithms are capable of computing complex scenarios, such as multi-objective optimization or game-theoretic analysis, they still operate exclusively on complete ADTs and do not provide support for localized analysis [10]. The literature also highlights the absence of flexible query languages, which limits partial analysis [4].

The study by Stefano Nicoletti and his colleagues highlighted the importance of using more complex queries for the analysis of Attack and Fault Trees beyond simple metric computation [33]. Their work introduced ATM (A Logic for Attack Tree Metrics), which allows specifying and checking structural and quantitative properties directly on ATs. This enables modeling what-if scenarios, quantification over classes of attacks, and reasoning about the success or failure of specific sub-attacks—capabilities that go well beyond traditional metric evaluation.

However, it is important to emphasize that existing tools and frameworks are not capable of such expressive analysis of ADTs, and as a result, they are unable to answer complex queries about the structure or behavior of subtrees, dependencies between specific

nodes, or the effects of localized changes. This scientific gap is implicitly highlighted in the literature, calling for the development of more flexible techniques that could support both global and local properties of ADTs [32].

1.5 Applicability of PRISM

To address the limitations of existing ADT analysis approaches, model checking techniques will be used, specifically the PRISM model checker. PRISM is a widely used tool for modeling and analyzing systems with probabilistic, nondeterministic, or real-time behavior. It is particularly suitable for ADT analysis due to its native support for quantitative analysis, probabilistic reasoning, and flexible specification languages such as PCTL and CSL.

Unlike other model checkers (like NuSMV or UPPAAL), PRISM is explicitly designed to handle probabilistic models and cost-based reasoning, which are essential when analyzing complex attack-defense scenarios involving uncertainty and resource allocation.

Several works have already demonstrated the use of model checkers for analyzing trees. Zaruhi Aslanyan and colleagues, in the work "Quantitative Verification and Synthesis of Attack-Defence Scenarios" [3], presented an extension of attack-defence trees to model temporal ordering of actions and explicit dependencies in attacker/defender strategies. Also, Yassmeen Elderhalli and colleagues, in the work "Formal Dynamic Fault Trees Analysis Using an Integration of Theorem Proving and Model Checking" [14], used model checking techniques to analyze Dynamic Fault Trees, demonstrating how formal verification can systematically identify system vulnerabilities and improve safety and reliability assessments.

Unlike traditional algorithms for quantitative analysis on ADTs, PRISM supports property-driven exploration of the model, allowing localized analysis of trees. This also enables simulation of what-if analysis, and the ability to define and evaluate complex security objectives without recomputing the entire tree from scratch. And of course, PRISM's specification languages make it possible to encode a wide range of quantitative and structural properties, including cost evaluation, success probability, and resource constraints.

1.6 Research Goal

The goal of this thesis is to address these limitations by developing a tool, that bridges the gap between high-level graphical ADT models and low-level formal representations suitable for in-depth formal analysis. The proposed tool is going to automate the transformation of ADTs into stochastic multiplayer game models, enabling expressive modeling of attacker–defender interactions, multi-metric evaluation, and formal strategy verification via the PRISM model checker [35].

This will be achieved through the development of a tool in a high-level programming language that automates the transformation of an XML file from ADTool into a PRISM model, which can then be used to perform global or localized analysis of the tree. Additionally, this tool will provide the capability to compute the Pareto Front for the ADT, ensuring a comprehensive view of the interaction between the attacker and the defender.

This scientific work consists of several main parts, including a detailed literature review related to ADT analysis, the design of the tool, the generation of the PRISM model, the algorithm for calculating the Pareto front, experiments on the computational performance of the tool, and a discussion on potential functional extensions.

Chapter 2

Background and related work

Understanding and assessing the security of complex systems requires robust modeling frameworks and systematic analysis techniques. This chapter provides an overview of Attack–Defense Trees (ADTs) as a foundational formalism for representing security scenarios, discusses established approaches for ADT analysis, and reviews existing tools used in the field. By highlighting both the strengths and limitations of current methods, this chapter sets the stage for the motivation and development of new, more expressive and verifiable solutions presented in the following sections.

2.1 Attack-Defense Trees

An Attack-Defense Tree is a formalism used to model the possible ways a system can be attacked and defended. Formally, an ADT is a node-labeled rooted tree, where the root represents the primary goal of the attacker, and the child nodes decompose this goal into sub-goals using logical operations (most common are AND and OR). Defensive nodes can be attached to attack nodes to represent countermeasures the defender may employ. The ADT structure is an extension of Attack Trees, representing the perspective of two players on system security [24].

Definition 2.1.1: Directed Acyclic Graph (DAG)

A *directed acyclic graph* is a pair $G = (V, E)$ where:

- V is a finite set of vertices (nodes),
- $E \subseteq V \times V$ is a set of directed edges,

such that there is no sequence of nodes $v_1, v_2, \ldots, v_k$ with $k \geq 2$ for which:

$$(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1) \in E$$

In other words, a DAG contains no directed cycles.

Definition 2.1.2: Attack-Defense Tree

An *Attack-Defense Tree (ADT)* is a labeled, rooted, directed acyclic graph:

$$T = (V, E, r, \mathit{type}, \mathit{role})$$

where:

- V is the set of nodes (vertices),
- $E \subseteq V \times V$ is the set of directed edges,
- $r \in V$ is the root node,
- *type* : $V \to \{\mathsf{AND}, \mathsf{OR}, \mathsf{INH}, \mathsf{LEAF}\}$ assigns to each node its structural type:
  - AND, OR, INH denote logical gates,
  - LEAF denotes a basic action (atomic attack or defense),
- *role* : $V \to \{\mathsf{att}, \mathsf{def}\}$ assigns each node its role in the tree:
  - att indicates attacker-controlled nodes,
  - def indicates defender-controlled nodes.

Constraint: Nodes of type LEAF must not have outgoing edges (i.e., they are terminal):

$$\forall v \in V,\ \mathit{type}(v) = \mathsf{LEAF} \Rightarrow \nexists v' \in V \text{ such that } (v, v') \in E$$

Definition 2.1.3: Basic Events

Let $T = (V, E, ...)$ *be an ADT, and* Let $v \in V$ be a node such that $\text{type}(v) = \texttt{LEAF}$. We call such a node a **basic event**.
A basic event is either:

- a **basic attacker event**, if $\text{role}(v) = \texttt{att}$,
- or a **basic defender event**, if $\text{role}(v) = \texttt{def}$.

These nodes correspond to the *atomic actions* available to the attacker and defender, and are represented as the leaves of the Attack-Defense Tree.

We define:

- $V_{\text{att}} = \{v \in V \mid \text{lab}(v) = \text{att}\}$ — the set of attack nodes, representing actions or goals of the attacker,
- $V_{\text{def}} = \{v \in V \mid \text{lab}(v) = \text{def}\}$ — the set of defense nodes, representing countermeasures or protective actions of the defender,
- $V_{\text{op}} = \{v \in V \mid \text{lab}(v) \in \text{AND}, \text{OR}, \text{INH}\}$ — the set of operator nodes.

Following [23], we define two functions over the set of nodes V:

- The function $\text{type} : V \to \{\text{AND}, \text{OR}, \text{INH}, \text{LEAF}\}$ assigns to each node its structural type, indicating whether the node is a logical gate or a basic event.
- The function $\text{role} : V \to \{\text{att}, \text{def}\}$ assigns to each node its player role: attacker (`att`) or defender (`def`).

In particular, the set of basic events (i.e., leaf nodes) is defined as:

$$V_{\text{leaf}} = \{v \in V \mid \text{type}(v) = \text{LEAF}\}$$

Each basic event corresponds to either an attacker or defender action:

$$\forall v \in V_{\text{leaf}}, \quad \text{role}(v) \in \{\text{att}, \text{def}\}$$

This separation between structural type and role allows representing both attacker and defender behavior in a unified tree structure.

We assume the following structural restrictions [10]:

- An attack-labeled gate can only have attack-labeled child nodes.
- A defense-labeled gate can only have defense-labeled child nodes.
- Inhibition gates may model conditional failure of an attack node if the associated defense is active.
- An inhibition gate must have exactly two children with different roles: one attack node and one defense node.

The tree alternates layers of attack and defense nodes, reflecting the interaction between the two players. Each internal node in the tree is associated with a logical operator (AND or OR).

Let $A = \{a_1, \ldots, a_n\}$ be the set of basic attack actions, and $D = \{d_1, \ldots, d_m\}$ be the set of basic defense actions. Let $\vec{a} \in \{0,1\}^n$ and $\vec{d} \in \{0,1\}^m$ be Boolean vectors representing which attacker and defender actions are active, respectively. Let V be the set of all nodes in the Attack-Defense Tree.

Definition 2.1.4: Structure Function

A structure function is a mapping

$$f : \{0,1\}^n \times \{0,1\}^m \times V \to \{0,1\}$$

such that $f(\vec{a}, \vec{d}, v)$ returns 1 if the goal or subgoal represented by node $v \in V$ is successfully achieved under attacker strategy $\vec{a}$ and defender strategy $\vec{d}$, and 0 otherwise.

The evaluation of the structure function f is defined recursively as follows:

- If v is a **leaf node** corresponding to attacker action a_i, then:

$$f(\vec{a}, \vec{d}, v) = \vec{a}[i]$$

- If v is a **leaf node** corresponding to defender action d_j, then:

$$f(\vec{a}, \vec{d}, v) = \neg \vec{d}[j]$$

- If v is an **AND-node** owned by the attacker, with children $v_1, \ldots, v_k$, then:

$$f(\vec{a}, \vec{d}, v) = \bigwedge_{i=1}^{k} f(\vec{a}, \vec{d}, v_i)$$

- If v is an **OR-node** owned by the attacker, with children $v_1, \ldots, v_k$, then:

$$f(\vec{a}, \vec{d}, v) = \bigvee_{i=1}^{k} f(\vec{a}, \vec{d}, v_i)$$

- If v is an **INHIBIT-node**, then its evaluation depends on the evaluation of two child nodes: the *target node* v_{target} and the *inhibiting node* $v_{\text{inhibitor}}$ (which can be either an attack or a defense subformula). The inhibit node is true if the target is true and the inhibitor is false:

$$f(\vec{a}, \vec{d}, v) = f(\vec{a}, \vec{d}, v_{\text{target}}) \wedge \neg f(\vec{a}, \vec{d}, v_{\text{inhibitor}})$$

This recursive function allows to compute the success or failure of any node in the tree, and can be used as a foundation for both global and localized analysis of ADTs.

Example 2.1.1: Attack-Defense Tree: Accessing a Vehicle

The ADT in Fig. 2.1 represents a scenario for gaining control over an electric vehicle. The *Access Vehicle* node is the top-level event and is modeled as an OR gate. The attacker has three possible attack vectors: *Steal Key, App Access, and Vulnerability Exploitation.* At the same time, the defender can apply a countermeasure against *App Access* by enforcing a *Strong Password.* However, the attacker may neutralize this defense by exploiting a vulnerability, represented by the *Vulnerability Exploitation* node.
There are two main attack scenarios available to the attacker: via *Steal Key* or via *App Access.* If the attacker chooses to steal the keys, they immediately gain control over the vehicle:

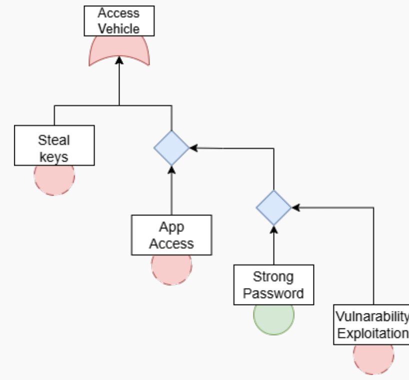

FIGURE 2.1: Attacke-Defense Tree, which describes the proccess of gaining control over a vehicle

$$A = \{a_1 = \text{Steal Key} = 1\},$$
$$D = \emptyset$$

$$\begin{aligned} f(\vec{a}, \vec{d}, \text{Access Vehicle}) &= f(\vec{a}, \vec{d}, \text{Steal Key}) \vee f(\vec{a}, \vec{d}, \text{INH}) \\ &= 1 \vee f(\vec{a}, \vec{d}, \text{INH}) \\ &= 1 \vee 0 = 1 \end{aligned}$$

If the attacker attempts to compromise the vehicle via App Access and the defender does not apply the Strong Password defense, then:

$$A = \{a_2 = \text{App Access} = 1\},$$
$$D = \emptyset$$

$$\begin{aligned} f(\vec{a}, \vec{d}, \text{Access Vehicle}) &= f(\vec{a}, \vec{d}, \text{Steal Key}) \vee f(\vec{a}, \vec{d}, \text{INH}) \\ &= 0 \vee \Big(f(\vec{a}, \vec{d}, \text{App Access}) \wedge \neg f(\vec{a}, \vec{d}, \text{Strong Password})\Big) \\ &= 0 \vee (1 \wedge \neg 0) \\ &= 0 \vee (1 \wedge 1) \\ &= 0 \vee 1 = 1 \end{aligned}$$

The next scenario assumes, that the attacker again chooses to attack *App Access* and the defender uses his countermeasure - *Strong Password*:

$$A = \{a_2 = \text{App Access} = 1\},$$
$$D = \{d_1 = \text{Strong Password} = 1\}$$

$$\begin{aligned}
f(\vec{a}, \vec{d}, \text{Access Vehicle}) &= f(\vec{a}, \vec{d}, \text{Steal Key}) \lor f(\vec{a}, \vec{d}, \text{INH}) \\
&= 0 \lor \Big(f(\vec{a}, \vec{d}, \text{App Access}) \land \neg f(\vec{a}, \vec{d}, \text{Strong Password})\Big) \\
&= 0 \lor (1 \land \neg 1) \\
&= 0 \lor (1 \land 0) \\
&= 0 \lor 0 = 0
\end{aligned}$$

The final scenario assumes that the attacker proceeds via App Access and also uses Vulnerability Exploitation, while the defender activates the Strong Password defense. In this case, the attack and defense vectors are defined as follows:

$$\begin{aligned}
A = \{&a_1 = \text{Steal Key}, \\
&a_2 = \text{App Access}, \\
&a_3 = \text{Vulnerability Exploitation}\}, \\
D = \{&d_1 = \text{Strong Password}\}
\end{aligned}$$

$$\begin{aligned}
f(\vec{a}, \vec{d}, \text{Access Vehicle}) &= f(\vec{a}, \vec{d}, \text{Steal Key}) \lor f(\vec{a}, \vec{d}, \text{INH}) \\
&= 0 \lor \Big(f(\vec{a}, \vec{d}, \text{App Access}) \land \neg f(\vec{a}, \vec{d}, \text{Strong Password})\Big) \\
&= 0 \lor \Big(1 \land \neg \Big(1 \land \neg f(\vec{a}, \vec{d}, \text{Vulnerability Exploitation})\Big)\Big) \\
&= 0 \lor (1 \land \neg (1 \land \neg 1)) \\
&= 0 \lor (1 \land \neg (1 \land 0)) \\
&= 0 \lor (1 \land \neg 0) \\
&= 0 \lor (1 \land 1) \\
&= 1
\end{aligned}$$

From these calculations, it is evident that the attacker is always able to gain control over the vehicle, except in the scenario where the attacker chooses to target the *App Access* node without performing *Vulnerability Exploitation*, and the defender applies their sole countermeasure - the *Strong Password*.

2.1.1 Quantitative analysis on ADTs

ADTs provide a well-defined formal semantics that supports both qualitative and quantitative analysis:

- Each leaf node is assigned a value for the attribute (cost).
- Internal nodes aggregate the values of their children according to the operator.

Each basic action in the ADT is associated with a non-negative cost representing required resources, effort, or any other relevant metric. The attack–defense tree structure T, together with the attacker and defender cost vectors

$$\vec{c}_{\text{att}} \in \mathbb{R}^n_{\geq 0} \quad \text{and} \quad \vec{c}_{\text{def}} \in \mathbb{R}^m_{\geq 0},$$

are part of the problem input. Here, $\vec{c}_{\text{att}}[i]$ denotes the cost of performing the i-th attacker action, and analogously for $\vec{c}_{\text{def}}[j]$.

Given these inputs, the players choose their respective strategies. Let

$$\vec{a} \in \{0,1\}^n \quad \text{and} \quad \vec{d} \in \{0,1\}^m$$

be Boolean activation vectors indicating which basic attacker and defender actions are selected (active) in a particular scenario. The vectors $\vec{a}$ and $\vec{d}$ represent the choices made by the attacker and defender, respectively.

Let $\vec{c}_{\text{att}}[i]$ represents the cost of the i-th attacker action, and similarly for the defender.

Definition 2.1.5: Cost Functions

The total cost of a strategy is defined as the dot product between the activation vector and the corresponding cost vector:

$$\text{Cost}_{\text{att}}(\vec{a}) = \sum_{i=1}^{n} \vec{a}[i] \cdot \vec{c}_{\text{att}}[i]$$

$$\text{Cost}_{\text{def}}(\vec{d}) = \sum_{j=1}^{m} \vec{d}[j] \cdot \vec{c}_{\text{def}}[j]$$

This formulation allows to evaluate the efficiency of each strategy in terms of required resources for both attacker and defender, and forms the basis for Pareto-optimal trade-off analysis.

It is worth noting that, in addition to the cost attribute, leaf nodes can be assigned various other attributes such as probability, time, skill level, and more. The general framework for handling these diverse metrics is based on the concept of semiring attribute domains [8]. A *semiring* is an algebraic structure consisting of a set equipped with two operations (often denoted as $\oplus$ and $\otimes$), which satisfy certain axioms such as associativity and distributivity. In the context of attack trees, semiring-based metrics allow for a unified and flexible way to compute a wide range of quantitative attributes.

The key advantage of this approach is that, as long as the attribute domain forms a semiring, the metric can be efficiently computed using a bottom-up algorithm that propagates values from the leaves to the root of the tree [29].

It is important to note that while semiring-based metrics provide a powerful and general framework, most existing tools do not support arbitrary semirings. For example, the widely used PRISM model checker is limited to handling real-valued costs and probabilities, and does not support general semiring attribute domains. This restricts its applicability for metrics that require more complex algebraic structures, such as those involving tuples, Pareto fronts, or non-numerical attributes. Due to this, in this thesis, the cost attribute is considered the primary metric, and all other attributes are beyond the scope of this work.

These semantics allow for alternating attack and defense nodes, effectively capturing realistic security scenarios and modeling strategic interactions between attackers and defenders.

Following [24], ADTs can be interpreted as a formal two-player game between the attacker and the defender. Each move corresponds to the choice of an attack or defense action at the corresponding level of the tree. This game-theoretic interpretation enables the application of formal verification and strategy synthesis techniques.

Example 2.1.2: Cost Analysis of Vehicle Access ADT

This example illustrates the application of cost-based analysis using the same vehicle access attack–defense tree from Fig. 2.1.The Cost aggregation can be seen on Fig. 2.2. The attacker can choose from three basic actions:

- a_1 = **Steal Key** (cost: 20),
- a_2 = **App Access** (cost: 10),
- a_3 = **Vulnerability Exploitation** (cost: 15).

The defender has one countermeasure:

- d_1 = **Strong Password** (cost: 5).

The cost vectors are:

$$\vec{c}_{\text{att}} = [20,\ 10,\ 15], \quad \vec{c}_{\text{def}} = [5]$$

We consider several attacker–defender strategies and compute the total cost for each side.

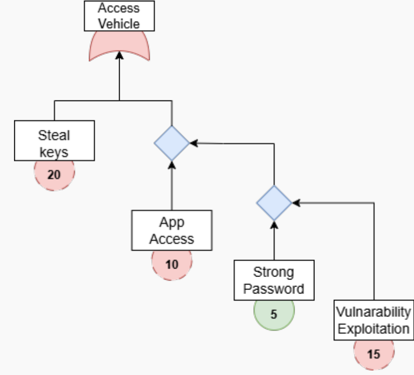

FIGURE 2.2: Attacke-Defense Tree, which describes the process of gaining control over a vehicle with costs

Scenario 1: Steal Key only

$$\vec{a} = [1,\ 0,\ 0], \quad \vec{d} = [0]$$

$$\text{Cost}_{\text{att}} = 1 \cdot 20 = \boxed{20}, \quad \text{Cost}_{\text{def}} = 0 \cdot 5 = \boxed{0}$$

Scenario 2: App Access only, no defense

$$\vec{a} = [0,\ 1,\ 0], \quad \vec{d} = [0]$$

$$\text{Cost}_{\text{att}} = 1 \cdot 10 = \boxed{10}, \quad \text{Cost}_{\text{def}} = 0 \cdot 5 = \boxed{0}$$

Scenario 3: App Access + Vulnerability Exploitation, defender applies Strong Password

$$\vec{a} = [0,\ 1,\ 1], \quad \vec{d} = [1]$$

$$\text{Cost}_{\text{att}} = 10 + 15 = \boxed{25}, \quad \text{Cost}_{\text{def}} = 1 \cdot 5 = \boxed{5}$$

Scenario 4: App Access only, defender applies Strong Password (attack fails)

$$\vec{a} = [0,\ 1,\ 0], \quad \vec{d} = [1]$$

$$\text{Cost}_{\text{att}} = 10, \quad \text{Cost}_{\text{def}} = 5$$

In this scenario, the attacker fails to gain access due to the activated countermeasure.

2.1.2 Pareto Front

The trade-off between the defender's and attacker's metric values can be analyzed via the Pareto front [12]. A solution $x^* \in X$ is called *Pareto optimal* if there is no other solution $x \in X$ that dominates x^*. Formally, for a multi-objective function $f : X \to \mathbb{R}^k$, a solution x^1 dominates x^2 (denoted $x^1 \prec x^2$) if

$$\forall j \in \{1, \ldots, k\} : f_j(x^1) \leq f_j(x^2) \quad \text{and} \quad \exists j : f_j(x^1) < f_j(x^2).$$

The *Pareto front* is the set of all non-dominated solutions:

$$PF = \{f(x) \mid x \in X, \nexists y \in X : f(y) \prec f(x)\}$$

In the ADT context, the Pareto front represents optimal trade-offs between a defender's and attacker's objectives. This trade-off of strategies can be evaluated by a multi-objective function

$$f : D \to \mathbb{R}^2,$$

where D is the set of all possible *defense vectors* $d \in D$. Each defense vector encodes a specific choice of active defensive actions. The objectives are defined as:

- **Attacker's objective**: $f_2(d) = -\text{cost}^{\min}_{\text{att}}(d)$ — the minimal cost that the attacker needs to succeed, given the defense vector d. It could be considered as maximizing attack success and minimize costs. That is, to carry out a successful attack on the system with minimal expenditure of resources.

- **Defender's objective**: $f_1(d) = \text{cost}_{\text{def}}(d)$ — the total cost incurred by the defender when using defense vector d. It could be considered as maximizing system security and minimize costs. That is, to guarantee system protection with minimal resource expenditure.

For more complete formal treatment, see [10, 3.4 Pareto Front].

The defender aims to minimize $f_1(d)$ (own cost), and maximize $f_2(d)$ (i.e., maximize the attacker's required effort). Therefore, Pareto-optimal defense strategies correspond to those vectors d for which no other vector d' exists such that:

$$f_1(d') \leq f_1(d) \quad \text{and} \quad f_2(d') \geq f_2(d),$$

with at least one inequality being strict.

A combination of attacker's and defender's actions is **Pareto Optimal** if there is no other solution $x \in X$ that reduces both player's costs simultaneously. So, formally it can be explained as

$$x^1 \prec x^2 \quad \text{if} \quad \forall j \in \{1, 2\} : f_j(x^1) \leq f_j(x^2) \quad \text{and} \quad \exists j : f_j(x^1) < f_j(x^2)$$

2.1.3 Attack Trees vs Attack-Defense Trees

In contrast, the basic concept of Attack Trees is to describe system failure states and the events leading to these failures. Each node in Attack Tree can have multiple children representing events that contribute to the failure of their parent. The "system" can refer to code, computer operations, travel plans, or any other domain where failure analysis is relevant.

The main difference between ADTs and standard Attack Trees lies in their context: ADTs explicitly model the interaction between attackers and defenders, including risks and counter-strategies. The main components of ADTs are attack nodes, which represent malicious actions or vulnerabilities, and defense nodes, which represent countermeasures. The addition of defense nodes distinguishes ADTs from Attack Trees, introducing both proactive and reactive elements to the model.

ADTs utilize logical operations such as AND and OR, similar to Attack Trees, allowing the hierarchical combination of attack and defense nodes to model complex interactions. This structure provides an organized and intuitive way to illustrate possible attack paths and corresponding defenses, making it easier for both technical teams and decision-makers to understand the relationships between vulnerabilities and mitigations.

Finally, ADTs reflect the dynamic nature of security risks and responses by combining proactive and reactive components. Logical operators such as AND and OR enable the modeling of real-world situations where multiple factors interact simultaneously, supporting a more nuanced understanding of system security.

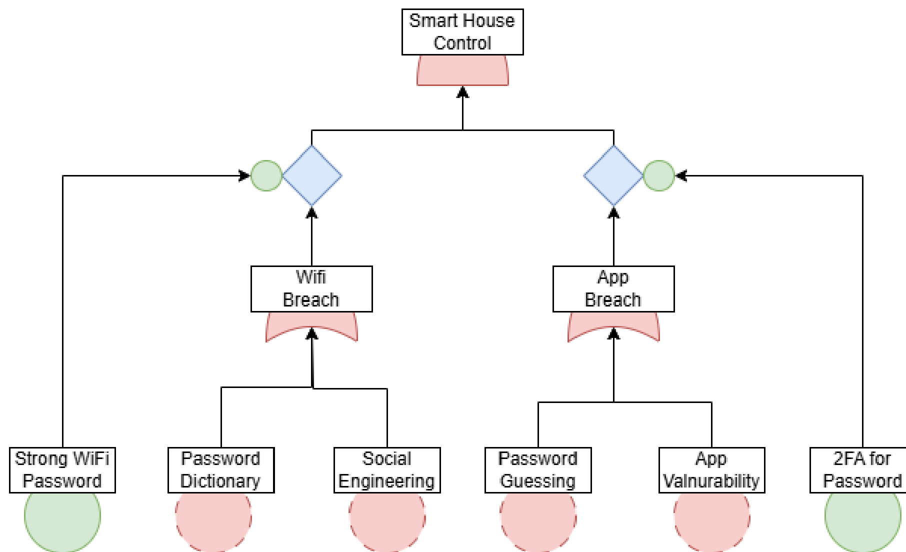

FIGURE 2.3: Trivial example of ADT structure, representing attack flow on Smart House Control System

Example 2.1.3: Smart House Control Scenario

To explain how ADTs work, the example in Figure [2.3] will be discussed. This Figure illustrates an Attack-Defense Tree for a smart house control system. The root node, labeled "Smart House Control," represents the ultimate goal of the attacker, which is to gain full control over the smart house system. This root node is connected to two primary attack paths: "WiFi Breach" and "App Breach." The "WiFi Breach" node models two possible attack methods: using a password dictionary to guess the WiFi password or employing social engineering tactics to deceive the user into revealing their credentials. Similarly, the "App Breach" node models two attack methods: attempting to guess the password for the smart house application or exploiting vulnerabilities in the app's security. Each of these attack nodes is connected to an OR gate, meaning that success requires any one of the child methods

to succeed.
Defensive nodes are also included in the tree to model countermeasures. The defensive node "Strong WiFi Password" is associated with the "WiFi Breach" node and reduces the likelihood of a successful attack by requiring a complex password for the WiFi network. Similarly, two-factor authentication for password (2FA for Password) node is associated with "App Breach" node and prevents the attack by this route.
Having established the structure and capabilities of Attack-Defense Trees through both formal description and practical example, we now turn to the analysis of these models.

2.2 Approaches for ADT analysis

Choosing suitable approaches that strike a compromise between precision, computational effectiveness, and relevance to actual security situations is necessary when analyzing ADTs, which can be evaluated using a variety of methods, each with unique benefits and drawbacks. These methods range from multi-criteria analysis, which uses Pareto front computation to evaluate trade-offs between attacker and defender strategies, to single-metric computations, which concentrate on evaluating qualities like cost or likelihood through bottom-up aggregation.

2.2.1 Existing ADT Analysis Tools

Currently, there are several existing tools that provide functionality for analyzing and visualizing ADTs. These tools are widely used in both commercial domains and academic research. They are worth considering, as they are direct counterparts to the tool being developed as part of this thesis. In the end of this section the comparitive analysis will be provided.

ADTool

This is one of the most well-known tools for constructing and performing quantitative analysis of ADTs. It provides functionality for visualizing ADTs, assigning attributes, and performing a bottom-up evaluation of a single attribute (this method will be discussed in more detail later, see Section **??**). Despite its user-friendly interface and wide adoption in academic circles [22], ADTool is limited to trees only (no support for DAGs), does not support strategic analysis, automatic verification, or export to formal models.

SecurITree

This is a commercial tool used in industry for modeling threats based on Attack Trees and Attack–Defense Trees. It supports DAG structures, attributes, probabilistic scenarios, economic calculations, and graphical visualization of threat scenarios [19]. However, unlike academic tools, SecurITree does not provide capabilities for formal verification, does not export models to verification tools, and does not support game-theoretic interpretation.

QUADTool

This is an academic development aimed at formalizing the analysis of ADTs [13]. It extends ADTool with the ability to automatically export models to PRISM and verify properties

using Markov Decision Process models. QUAD supports quantitative analysis with multiple attributes and scenarios, but it does not model the interaction of two participants as a strategic process and has limited support for DAG structures.

Comparison of tools

Table 2.1: Comparison of ADT Analysis Tools

Feature	**ADTool**	**SecurITree**	**QUADTool**
Visual ADT modeling	✓	✓	✓ (via ADTool)
Support for Attack-Defense Trees	✓	✓	✓
Support for DAG structures	✗ (trees only)	✓	limited
Custom attributes (cost, probability, etc.)	✓	✓	✓
Bottom-up attribute evaluation	✓	✓	✓
Formal verification support	✗	✗	✓ (via PRISM)
Export to PRISM-compatible model	✗	✗	✓
Game-theoretic model (2-player interaction)	✗	✗	✗
Strategy analysis and trade-off exploration	✗	partial(manual)	partial, via MDP
Multi-objective analysis (e.g., Pareto front)	✗	partial	partial
Open-source	✓	✗ (proprietary)	✓
Extensibility / integration capabilities	partial(via XML)	✗	✓ (PRISM input)
Support for formal logics (e.g., PCTL, CSL)	✗	✗	✓

As shown in Table 2.1, existing tools for Attack-Defense Tree analysis exhibit a variety of strengths, yet none of them fully address the needs of expressive, formally verifiable, and strategy-aware modeling. While ADTool and SecurITree provide accessible graphical modeling environments with basic support for quantitative analysis, they lack capabilities for formal verification or game-theoretic reasoning. On the other hand, QUADTool offers partial support for formal model generation and integration with the PRISM model checker, but does not natively support DAG structures or two-player game semantics.

These findings point to a weakness in the current tool support: there aren't many integrated frameworks that combine multi-objective reasoning, formal strategy analysis, scalable verification, and flexible modeling of complicated ADT structures. This encourages the creation of tools meant to close these gaps.

Before introducing our approach, the following sections provide a deeper look into

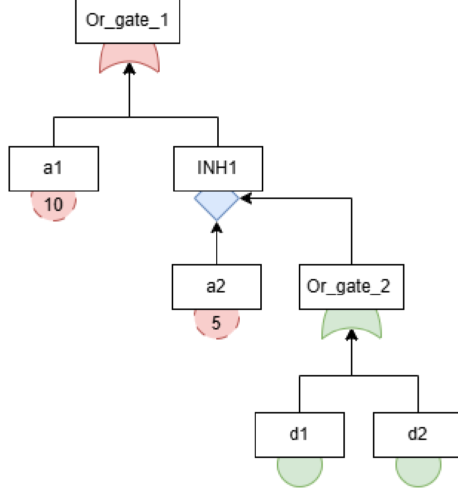

FIGURE 2.4: Example ADT with attacker costs

the computational foundations of ADT analysis, starting with the most basic form of evaluation: single metric computation.

2.2.2 Single metric computation

Single metric computation in the context of ADTs refers to the quantitative evaluation of a security scenario using a single attribute—such as attacker cost, probability of success, or required effort—focused on one of the parties involved (typically the attacker). This process involves assigning specific values to the basic actions, or leaf nodes, within the ADT and systematically aggregating these values through the tree's structure to derive an overall assessment from the perspective of that party [24].

Bottom-Up method

Formally, let $cost_{att} : V_{att} \to \mathbb{R}_{\geq 0}$ is a function that assigns costs to the attacker nodes. Then, for an internal attacker node with children $v_1, ..., v_n$, the cost aggregation function would look like this [27]:

Definition 2.2.1: Attacker cost aggregation function

Let $\text{cost}_{att} : V \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ be the function that assigns a cost to each node $v \in V$, reflecting the minimal cost for the attacker to succeed via this node. The aggregation is defined recursively as follows:

$$\text{cost}_{att}(v) = \begin{cases} \min_i \text{cost}_{att}(v_i), & \text{if } \mathsf{owner}(v) = \mathsf{att},\ \text{lab}(v) = \texttt{OR} \\ \sum_i \text{cost}_{att}(v_i), & \text{if } \mathsf{owner}(v) = \mathsf{att},\ \text{lab}(v) \in \{\texttt{AND}, \texttt{INH}\} \\ \sum_i \text{cost}_{att}(v_i), & \text{if } \mathsf{owner}(v) = \mathsf{def},\ \text{lab}(v) = \texttt{OR} \\ \min_i \text{cost}_{att}(v_i), & \text{if } \mathsf{owner}(v) = \mathsf{def},\ \text{lab}(v) \in \{\texttt{AND}, \texttt{INH}\} \\ \infty, & \text{if } v \text{ is a basic defender node} \end{cases}$$

Example 2.2.1: Cost computation

Figure 2.4 presents an ADT consisting of an attacker OR gate (`or_gate_1`), two basic attacker events (`a1` and `a2` with costs 10 and 5, respectively), an inhibition gate (`INH1`), a defender OR gate (`or_gate_2`), and two defender nodes — `d1` and `d2`.

Based on Definition 2.2.1, the attacker cost aggregation for `d1` and `d2` is ∞, since they are basic defender events. For `or_gate_2`, we apply the cost aggregation rule for an OR gate owned by the defender: $\infty + \infty = \infty$.

For `INH1`, we apply the aggregation rule for an inhibition gate owned by the attacker:

$$\text{cost}(\texttt{or_gate_2}) + \text{cost}(\texttt{a2}) = \infty + 5 = \infty.$$

For `or_gate_1`, we apply the rule for an OR gate owned by the attacker:

$$\min(\text{cost}(\texttt{INH1}), \text{cost}(\texttt{a1})) = \min(\infty, 10) = 10.$$

Thus, the cost for successfully attacking the system shown in Figure 2.4 is 10.

This example demonstrates the Bottom-Up algorithm. A key limitation of this approach is that it effectively incorporates information for only one party (e.g., the attacker), without simultaneously considering the impact or costs for the defender. As a result, single metric computation may not fully capture the interplay between attack and defense strategies within the scenario.

Clone Deletion Method

By detecting and combining redundant attack operations prior to attribute calculation, the Clone Deletion technique tackles the issue of repeated labels in ADTs [27]. By making sure that every distinct action is taken into account just once, this strategy prevents double-counting, in contrast to the bottom-up approach. As a result, calculated measures like cost and success likelihood are more accurate. However, if repeated labels indicate semantically separate actions in different contexts, the method may result in improper merging. Additionally, the pre-processing step adds computing complexity, particularly for large ADTs.

2.2.3 Pareto fronts between multiple metrics on the same party

For a single party Pareto front analysis in ADTs enables the assessment of trade-offs between competing metrics. This method identifies non-dominated strategies when enhancing one metric hurts another, in contrast to standard approaches that maximize a single criterion (e.g., minimizing cost) [12]. For example, one attack can be inexpensive but ineffective, whereas another might be expensive but very effective. Such ideal trade-offs are captured by the Pareto front, which provides a more sophisticated perspective on choosing a security plan.

Example 2.2.2: Pareto Front for Attacker's Strategies

Consider the following attacker strategies evaluated using two metrics: *Cost* and *Success Probability* in the context of cost minimization and probability maximization.

Strategy	Cost	Success Probability
A1 + A2	30	0.9
A3	15	0.4
A2 + A3	35	0.95

To construct the Pareto front, we discard dominated strategies:

- **A1 + A2** dominates **A3**: although more costly, it offers a significantly higher success probability.
- **A2 + A3** dominates **A1 + A2**: slightly more expensive, but achieves better results.

The remaining non-dominated strategies are:

- **A3** with (Cost = 15, Probability = 0.4)
- **A2 + A3** with (Cost = 35, Probability = 0.95)

These points form the **Pareto front** — a set of strategies where no single option is strictly better in both objectives.

Bottom-Up Method

As previously discussed, bottom-up evaluation is one of the most scalable and widely used approaches for quantitative analysis over ADTs. In the context of multi-metric evaluation, Pareto fronts can be naturally derived from semiring-based formulations [5]. Since most bottom-up techniques operate over a semiring structure (e.g., using min, $+$, or $\times$ as semiring operations), the same methodology can be extended to multi-objective settings by evaluating multiple such structures in parallel.

This allows computing Pareto-based optimization over several qualities simultaneously, including cost, time, and success probability [10]. Bottom-up algorithms remain effective even in this context, as they traverse the tree from leaves to root in linear time with respect to the number of nodes and edges, i.e., $O(|N| + |E|)$ [15].

Integer Linear Programming

Integer Linear Programming (ILP) has been effectively applied to compute Pareto fronts for standard attack trees, particularly in DAG-structured cases where node sharing makes bottom-up techniques imprecise due to repetitive counting [30]. Although the approach in [30] does not consider defense nodes, it can be naturally extended to ADTs by incorporating defense-related constraints.

ILP is well-suited for precise multi-metric evaluation, since—unlike tree-based methods—it allows formal modeling of dependencies and restrictions arising from shared substructures and multiple parent nodes. Furthermore, ILP formulations support bi-objective optimization, making it possible to solve key ADT evaluation problems involving trade-offs between attacker and defender strategies:

- DgC: Most damaging attack within a cost budget.
- CgD: Cheapest attack achieving minimum damage.

- CDPF: Full cost-damage Pareto front.

However, because solving several ILP subproblems might result in exponential runtimes, this method is computationally costly, particularly for large ADTs. [30] states that the general cost-damage problem is NP-complete.

2.2.4 Pareto fronts between attacker and defender

In cybersecurity, Pareto fronts are instrumental in analyzing the trade-offs between attackers and defenders. A Pareto front represents a set of strategies where improving one objective (e.g., enhancing security measures) cannot occur without compromising another 2.1.2 (e.g., increasing costs). This concept aids both attackers and defenders in identifying optimal strategies that balance multiple objectives. Attackers aim to maximize impact while minimizing resources and exposure, while defenders strive to minimize risks and costs while maximizing system resilience. This interplay between attacker and defender strategies can also be examined through combined Pareto fronts. This approach models the dynamic interactions between both parties, facilitating the identification of equilibrium points where neither can unilaterally improve their position. Such analyses are crucial for understanding the balance of power in cybersecurity scenarios and for developing robust defense mechanisms. Pareto Front calculation example you can see on Fig. 2.5.

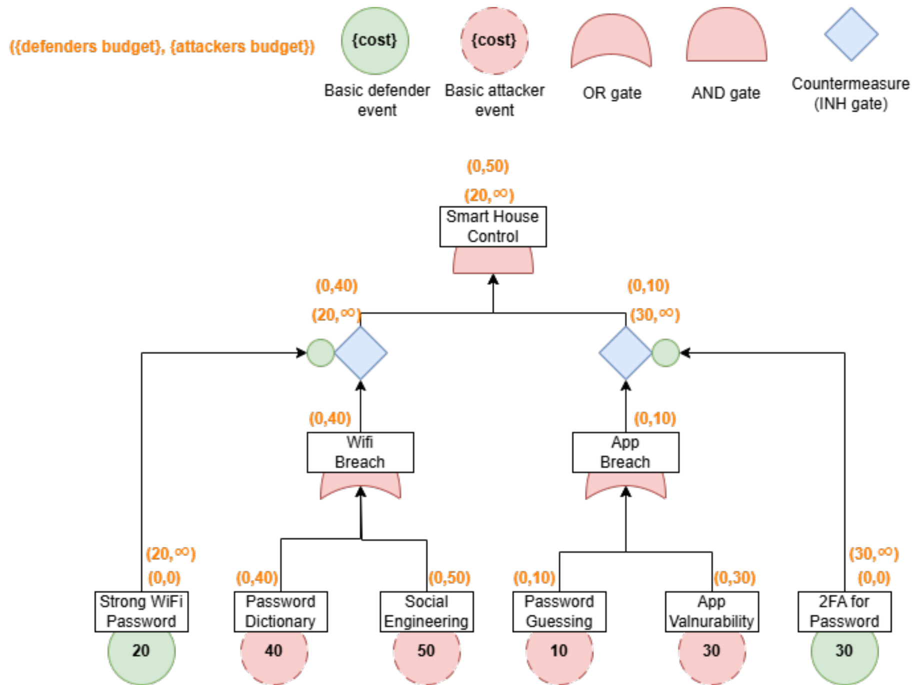

FIGURE 2.5: Pareto Front calculation for 2.3 with assigned costs

The given ADT (Fig.2.5) describes a system called the Smart House Control System. From the diagram, it is clear that the top node (i.e., the *Smart House Control* node) is an AND gate, which means that both the *WiFi Breach* node and the *App Breach* node must be compromised. The minimal cost required to breach both nodes corresponds to the set of attacker actions *{Password Dictionary, Password Guessing}*, which sums up to 50. The plot in Fig. 2.6 shows a point at 50 on the X-axis (Attacker cost), indicating that this is the minimum cost required to compromise the tree.

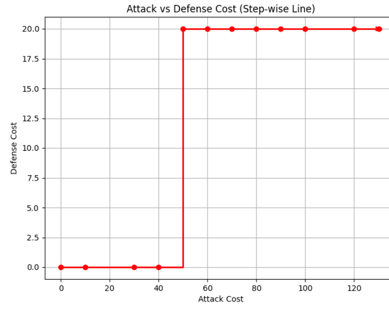

FIGURE 2.6: Pareto Front plot for 2.5

However, the defender only needs to apply one of the actions (*Strong WiFi Password* or *2FA for Password*) to guarantee protection of the tree, since, as mentioned earlier, the *Smart House Control* node is an AND gate, and preventing even one of the subgoals (*WiFi Breach* or *App Breach*) protects the entire tree. Therefore, in the context of minimizing resources while maximizing effectiveness, the defender chooses the cheaper action — *Strong WiFi Password*. As shown on the plot in Fig. 2.6, at a defender budget (Y-axis) of 20, the attacker cannot reach the top node under any scenario.

Bottom-Up Method

As previously introduced, the bottom-up algorithm evaluates ADTs recursively from leaves to the root by combining values based on logical node semantics. This method naturally extends to multi-objective Pareto front computation [10]. In this case, instead of aggregating a single value, each node propagates a set of non-dominated points representing trade-offs between attacker and defender objectives (e.g., cost vs. success probability).

The overall structure of the algorithm remains the same: local Pareto fronts are computed at leaves, recursively combined through AND/OR logic, and filtered to retain only Pareto-optimal strategies. The final Pareto front at the root node represents the best trade-offs from the attacker's or defender's perspective.

While efficient for tree-structured ADTs (running in linear time $O(|N|+|E|)$ [10]), this method does not handle node sharing in DAGs, where multiple parent dependencies can lead to overcounting. In such cases, ILP-based methods are more appropriate.

Biobjective Integer Linear Programming

A sophisticated optimization method called Biobjective Integer Linear Programming (BILP) is applied to Pareto front computing in DAG-structured ADTs [11]. By simultaneously optimizing two conflicting objectives—usually the attacker's cost versus the defender's cost—it goes beyond conventional ILP techniques. Because of this, BILP is especially effective in modeling security trade-offs, where it is necessary to take into account both attack and defense tactics in an organized manner.

In contrast to bottom-up algorithms, which work well only for tree-structured ADTs, BILP is effective for DAG-structured ADTs (i.e., ADTs where attack steps share common subcomponents). This allows for precise modeling of complex attack-defense interactions, avoiding double-counting issues present in naive bottom-up approaches [10].

Bi-objective integer linear programming (BILP) problems are NP-hard, which means that their computational complexity increases significantly as the number of attack-defense

actions grows. In the worst-case scenario, the time complexity to solve a BILP problem is given by

$$O(2^{|D|} \cdot n)$$

, where $|D|$ represents the number of defense actions and n denotes the number of nodes in the attack-defense tree. This exponential growth in computation time makes BILP impractical for very large security models, especially when the number of defensive actions is high. To address this limitation, heuristic methods and approximation techniques are often used to improve scalability while maintaining solution quality. In addition, most BILP-based formulations model costs as linear functions over Boolean decision variables. This representation is suitable for defender costs, where each enabled defense action contributes a fixed amount to the total cost. However, as shown by Copae et al. [10], attacker cost computation is often more complex due to shared nodes and nonlinear dependencies between attack steps. In such scenarios, the additive cost assumption may lead to inaccuracies.

Moreover, many real-world security scenarios exhibit nonlinear effects, such as compounding probabilities of success, interdependent actions, or adaptive adversarial behavior. These cases may require nonlinear optimization or stochastic modeling techniques to capture system behavior more accurately.

Binary Decision Diagrams based algorithms

Algorithms based on Binary Decision Diagrams (BDDs) compress huge Boolean expressions into a graph-based form, offering an effective method for evaluating DAG-structured ADTs [10]. These techniques perform exceptionally well when dealing with shared subtrees, a situation in which conventional bottom-up approaches fall short because of needless computations.

Unlike bottom-up approaches, which double-count nodes in DAG-based ADTs, BDD-based algorithms compress equivalent attack paths into a canonical Boolean function. This eliminates redundancy, making BDDs highly efficient for attack-defense scenarios with shared dependencies.

Copae et al. [10] demonstrate that BDD-based methods significantly outperform bottom-up and ILP-based approaches on DAG-structured ADTs, both in terms of runtime and memory consumption. In addition, BDD-based techniques allow for the efficient computation of Pareto fronts, where multiple security attributes must be optimized simultaneously. Unlike ILP-based methods, which require explicit constraint modeling, BDDs inherently support multi-objective analysis through symbolic Boolean function manipulations.

Despite their advantages, BDD-based algorithms face several challenges when applied to large-scale security models. Although BDDs reduce redundancy by canonicalizing Boolean functions, their size can still grow exponentially in certain ADT structures — particularly when the number of variables is large or when attack steps exhibit complex interdependencies.

A key limitation of BDDs lies in their sensitivity to variable ordering. An inefficient ordering can lead to a significant blow-up in size, negating their computational advantages. Finding an optimal variable order is an NP-hard problem, and thus heuristic strategies, such as sifting, are commonly used to approximate good orderings [7].

While such heuristics have proven effective in many general-purpose applications, they have not yet been explored specifically in the context of Attack–Defense Trees. This remains an open research direction, especially for DAG-structured ADTs where node sharing intensifies the ordering problem.

2.3 Shortcomings in Existing Approaches

While current ADT analysis techniques—such as single metric computations, Pareto front analyses, and methods based on Binary Decision Diagrams and Integer Linear Programming — provide a solid foundation for cybersecurity assessment, they exhibit significant limitations that hinder their effectiveness in real-world applications.

A primary shortcoming is the limited expressiveness of these approaches. Existing methods often assume static attacker-defender interactions, failing to capture the dynamic and adaptive nature of real-world cybersecurity scenarios where both attackers and defenders continuously adjust their strategies. This lack of expressiveness restricts the ability to model complex, evolving threats and nuanced defensive responses.

Moreover, formal verification is insufficiently addressed in most current frameworks. The absence of rigorous, formal mechanisms for verifying the correctness and completeness of security models means that critical vulnerabilities or strategic oversights may go undetected. This undermines confidence in the results of ADT-based analyses, especially when used to inform high-stakes security decisions.

The goal of this research is to overcome these critical limitations by enhancing the expressiveness and formal verification capabilities of ADT analysis. By transforming Attack-Defense Trees into two-player game automata and integrating probabilistic and scalable modeling tools with formal game-theoretic principles, this work aims to achieve greater dynamic flexibility, strategic realism, and practical reliability in cybersecurity risk assessment.

Chapter 3

Tool overview

To address the identified shortcomings of existing ADT analysis methods, this research leverages a combination of established tools. The foundation consists of ADTool, a tool for modeling Attack-Defense Trees, and the PRISM model checker, which supports stochastic multiplayer games (SMGs).

ADTool is a dedicated platform for constructing, visualizing, and performing basic quantitative analysis of ADTs. It provides a user-friendly interface for building complex attack–defense scenarios, assigning attributes (such as cost, probability, or effort) to nodes, and executing standard bottom-up computations. As highlighted by Kordy et al. [22], ADTool has become a de facto standard for ADT modeling in both academic and industrial contexts due to its accessibility and support for a range of security metrics. However, despite its strengths, ADTool is limited in terms of expressiveness: it primarily supports single-metric or simple multi-metric analysis and does not natively enable advanced trade-off exploration or strategic reasoning between attackers and defenders.

3.1 Choice for model checkers

Model checkers are essential in the process of converting Attack-Defense Trees into automata for two-player games and evaluating the generated models. These methods thoroughly examine potential states and transitions, enabling the formal proof of system attributes. PRISM, Uppaal, and NuSMV are a few of the commonly utilized tools. Although each offers special qualities, PRISM is clearly the best option for this study.

Comparison of Model Checkers

Criterion	PRISM	Uppaal	NuSMV
Support for Probabilistic Models	Fully supports probabilistic models, including discrete and continuous Markov chains, Markov Decision Processes, and stochastic multi-player games [35].	Focused on real-time systems; limited support for probabilistic models. [16]	No native support for probabilistic systems; focuses on Boolean models. [9]

Game-Theoretic Capabilities	Provides extensions for analyzing stochastic games, including computation of Nash equilibria and optimal strategies.[35]	Does not support game-theoretic analysis.[16]	Does not support game-theoretic analysis.[9]
Property Specification	Utilizes probabilistic temporal logics such as PCTL and CSL for expressing complex properties.[28]	Uses TCTL tailored for specifying timing properties in real-time systems.[16]	Supports CTL and LTL for Boolean state exploration.[9]
Robust Tool Ecosystem	Includes simulation, statistical model checking, and extensive integration capabilities. [35]	Primarily focused on real-time system simulation; limited statistical capabilities.[16]	Offers strong symbolic verification but lacks simulation and statistical analysis tools.[9]
Validation in Research	Extensively used in cybersecurity, including security protocol analysis and probabilistic program evaluation.[34]	Widely used for real-time systems, especially in embedded and control systems.[16]	Primarily applied in hardware verification and Boolean logic models.[9]
Community and Documentation	Large community with extensive documentation, case studies, and active development.[35]	Well-documented but primarily focused on real-time and embedded systems.[16]	Established community with a narrower focus on hardware verification.[9]

Table 3.1: Comparison of Model Checkers: PRISM, Uppaal, and NuSMV

It is evident that PRISM performs exceptionally well in fields that call for game-theoretic analysis and probabilistic modeling. It is perfect for situations involving dynamic attacker-defender interactions since it can describe stochastic multi-player games and Markov Decision Processes. Furthermore, complicated features can be expressed using its probabilistic temporal logics, like PCTL and CSL, which is crucial for cybersecurity research.

Limitations for Uppaal and NuSMV:

- **Uppaal:** While Uppaal is excellent for analyzing real-time systems with strict timing constraints, it has limited support for probabilistic systems and lacks game-theoretic capabilities. This makes it less suitable for the stochastic and adversarial nature of ADT scenarios.

- **NuSMV:** Known for its symbolic verification of Boolean models, NuSMV is effective in hardware verification and finite automata. However, it does not support probabilistic modeling or game-theoretic analysis, which limits its applicability in this research.

This comparison indicates that the best tool for converting ADTs into automata for two-player games and examining their behavior is PRISM. It is essential to this research because of its broad application in cybersecurity, game-theoretic capabilities, and substantial support for probabilistic models.

Chapter 4

ADTransformer: Tool Design and Implementation

As introduced in section 2.3, contemporary approaches dealing with Attack-Defense Trees lack sufficient expressiveness and automation, and are not able to support full-fledge risk and mitigation evaluation in cybersecurity. I developed the ADTransformer to overcome these limitations by providing a tool that automatically transforms ADTs into formal game automata and supports advanced analysis of trade-offs between attackers and defenders. This section discusses the design of ADTransformer, its design choices and the underlying principles behind its development. Check out the project on GitHub.[1]

4.1 Tool Architecture

This section describes the architecture (Fig. 4.1) for the ADTransformer tool that was built to support and automate the analysis of Attack-Defense Trees (ADTs) by translating them into a formal model for the evaluation of strategies. It is developed following the modular design pattern, so that each module looks to have a well hidden responsibility and communicate through a central orchestrator: The Command Executor.

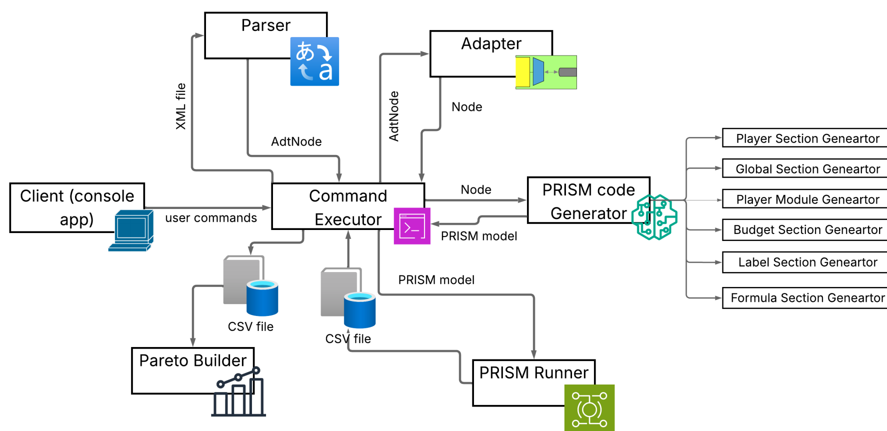

FIGURE 4.1: ADTransformer design schema

The high-level flow goes like this: It starts with the Client (Console Application). The tool communicates with the users by commands, the commands are sent to the Command

[1] https://github.com/AplyQ8/ADTransformer

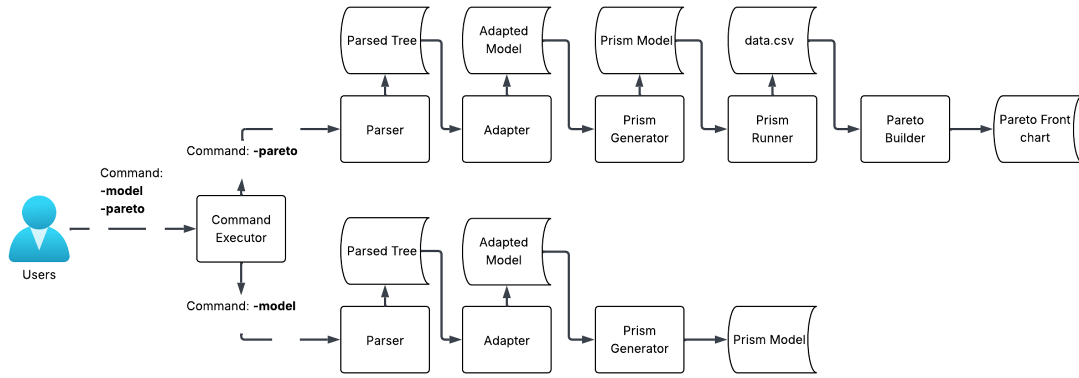

FIGURE 4.2: ADTransformer Data Flow Diagram

Executor. This is the central coordinator that manages interactions between all components.

The architecture consists of the following modules:

- **Parser** - This module processes the input XML from ADTool file describing the ADT and converts it into an internal tree structure.
- **Adapter** - The Adapter takes the parsed tree and transforms it into a format suitable for model generation in PRISM.
- **Prism Code Generator** - Based on the adapted structure, this component generates the corresponding PRISM model that formally represents the ADT.
- **Prism Runner** - This module invokes the PRISM model checker with the generated model and given properties. It verifies the model and outputs results to a data.csv file, representing Pareto Front points.
- **Pareto Builder** - Once the verification is complete, the ParetoBuilder uses the data in data.csv to construct a visual representation of trade-offs using a Python script.

Although the modules are logically connected (Parser → Adapter → PrismCodeGenerator → PrismRunner → ParetoBuilder), they are not tightly coupled. Rather, each module sends its output to the Command Executor, which then passes the result to the next item in the processing chain. This decoupling of concerns gives a neat, easy-to-maintain design and allows individual modules to be exchanged or upgraded without affecting the remainder of the system. The tool's architecture is illustrated in the diagram [Fig. 4.1], which emphasizes the pivotal position of the Command Executor and the flow of data between modules.

4.2 ADTool to PRISM: Transformation Workflow

This section is dedicated to the process of transforming an XML ADTool output file, which describes an ADT, into the corresponding PRISM model. It covers the structure of the resulting PRISM code, the processing of the XML file, the code generation steps, and presents a visual example of the final output.

4.2.1 PRISM code structure

A PRISM model can be divided into logical "blocks" 4.3, which allows the code to be generated sequentially.

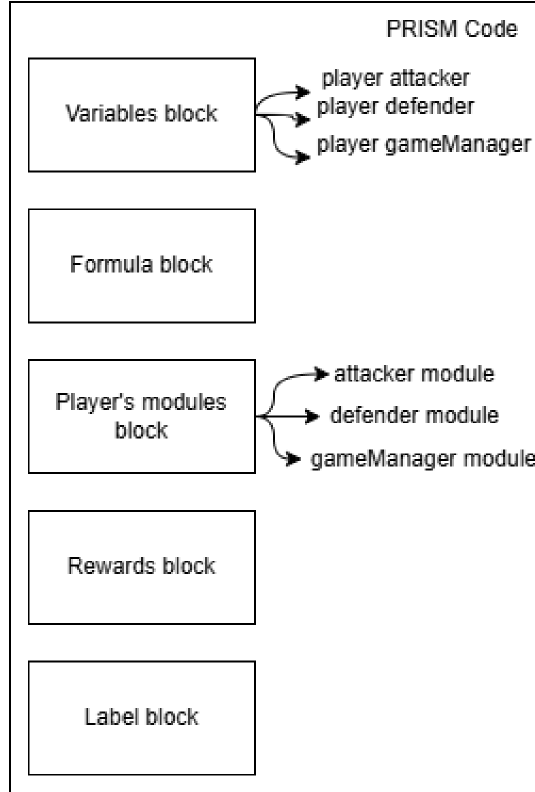

FIGURE 4.3: PRISM model structure schema

Variable block is a section of the code where all global variables are stored, including the players (attacker, defender, and game manager). In addition to the players, this block contains the turn variable for controlling the turn-based flow and a boolean variable top_event, which serves as a flag indicating whether the final goal has been reached. This block also includes variables representing the basic events of the players and their states: a value of 0 indicates an inactive event, while 1 denotes an activated one.

Formula block contains the formulas that define the gates of the tree. These formulas are nested within each other, which enables PRISM to perform state computations more efficiently.

Players Module Block declares the modules for the players. These modules allow players to choose actions during their respective turns and determine the attack or defense vector (in the case of attacker or defender turns), as well as compute the outcome of the game (in the case of the game manager's turn).

The reward block defines rewards for the players. Each action (basic event) performed by a player incurs a specific cost. Therefore, this block plays a key role in enabling a more detailed cost-based analysis.

Label block contains flags used to determine the current state of the tree (e.g., end_game, deadlock, etc.). It is also required for the further verification of the tree model.

4.2.2 Parser

The transformation process in ADTransformer does not begin directly with the PRISM Code Generator. Instead, it starts with an intermediate component — the Adapter — which plays a crucial role in preparing the data for code generation. Specifically, the Adapter takes the output of the XML Parser, which reads ADTs exported from ADTool, and converts it into a more flexible internal representation that the PRISM Code Generator can understand and operate on.

This intermediate layer is necessary because, despite the advantages of ADTool, it has a major limitation: it only supports strictly tree-structured Attack-Defense Trees. This constraint makes it difficult to represent more complex DAG-structured ADTs, where identical subgoals or subtrees can be reused across the structure.

To overcome this, the Adapter detects identical node names and automatically merges them into shared nodes, as it is shown on Fig. 4.4, effectively transforming the input into a DAG-like structure. This enhancement allows the resulting model to more accurately reflect real-world attack scenarios. As discussed in Section **??**, DAG-structured ADTs allow for substructure sharing, which makes them a better fit for modeling repeated attack patterns and dependencies in complex systems.

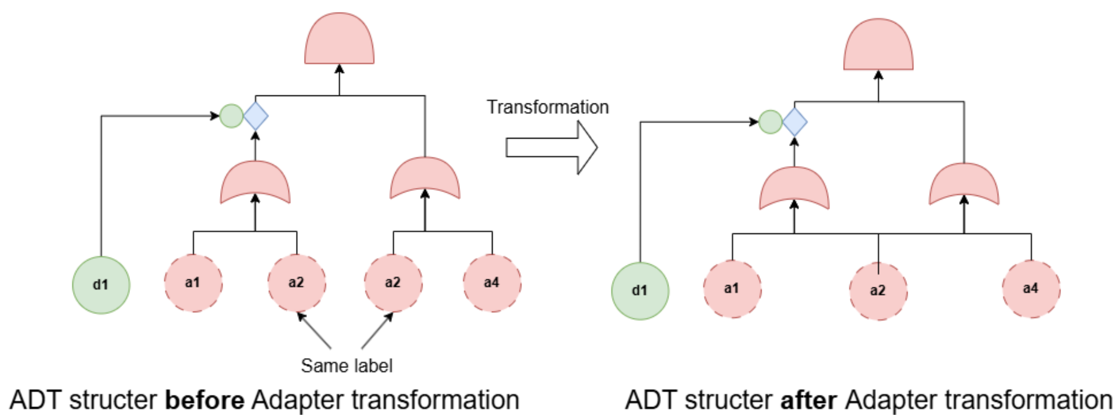

FIGURE 4.4: Adapter transformation process

The resulting internal structure is then passed to the PRISM Code Generator, which transforms it into a formal model suitable for probabilistic analysis.

4.2.3 PRISM Model Generation

At the heart of the PRISM code generation process lies a structured approach centered around the Node class 8.1, which acts as the core abstraction for representing ADT nodes. This class, along with a modular section-based generator system, enables flexible and extensible generation of PRISM code.

The Node class is the fundamental unit of the ADT structure in this tool. Each Node instance includes:

- **Id, Label**: Unique identifier and human-readable name.
- **GateType**: Optional enum for AND/OR gate semantics.
- **Owner**: An optional enum specifying attacker or defender ownership.

- **Cost**: Optional integer representing node cost (only for basic attacker/defender events).
- **Children**: A list of child Node objects, enabling tree (or DAG) structure.
- **IsActionNode, IsAttackerNode, IsDefenderNode, IsLeaf**: Flags used during code logic to conditionally trigger behavior.

This class serves as the data model that is consumed by each section generator to produce its portion of the PRISM model. As can be seen, the **GateType** can only be either AND or OR. This is due to the fact that, in the original ADTool structure, each node is assigned one of these two types: AND or OR. The presence of a countermeasure is indicated by a Boolean flag called "switch role" (set to yes for a countermeasure, no for a regular child node). Countermeasures are handled in the final PRISM model through formulas, described in 4.2.3, which are generated by a separate handler. This handler transforms the Node structure into a dedicated structure used as a basis for formula generation.

Lets assume we have DAG-structured ADT on Fig 4.5.

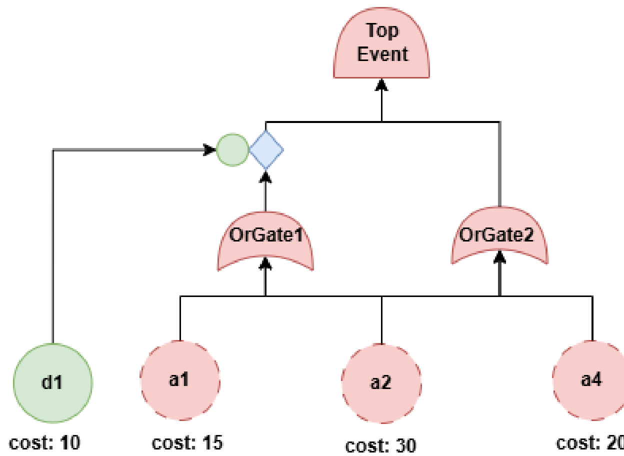

FIGURE 4.5: Example DAG-structured ADT with costs and labels

After adaptation process the node labeld as **"a2"** will be contained in the program memory as follows:

- **Label**: a2
- **GateType**: null
- **Owner**: Attacker
- **Cost**: 30
- **Children**: null
- **IsLeaf**: true
- **IsActionNode**: true
- **IsAttackerNode**: true

And for example **OrGate1** and **OrGate2** will differ from **a2** in the way that their **GateType** will be "OR" and they will have references on their children (for **OrGate1** it is **a1** and **a2**). Then, based on this node structure, code is generated.

The PrismCodeComposer class is the entry point for the transformation process. It receives a collection of Node instances and invokes a set of modular components called section generators, each responsible for a specific block of the final PRISM model.

Variable Block and Variable Generation Logic

Each of these blocks on Fig. 4.3 is generated by a corresponding section generator class. These generators operate independently but rely on a shared data structure — the Node class — which represents the ADT nodes. The Variable Block is responsible for declaring all relevant variables, constants, and modules needed to represent the attacker and defender logic in PRISM. This block is generated by such classes as **PlayerSectionGenerator**, **BudgetSectionGenerator**, **GlobalSectionGenerator** etc. These generators iterate through the ADT node structure and generate code accordingly. For example, For each attacker or defender node, an event is generated with a unique name composed of its label and id to avoid conflicts.

Formula Block and Formula Generation Logic

The **Formula Block** in the PRISM code generation process plays a central role in abstracting and expressing the logical relationships between nodes in the ADT. The generation of formulas is handled by the `FormulasSectionGenerator`, which delegates the actual construction of formulas to a singleton class called `FormulaTreeBuilder`. This design ensures that formulas are created only once and that shared expressions across the model are reused efficiently.

The core idea behind formula generation is to represent the triggered state of a node (whether an attack or defense node is considered active) as a Boolean formula in the PRISM language. The formulas are constructed recursively, starting from the root nodes of the ADT and walking through all children, while memoizing already visited nodes to avoid redundant computation and to prevent infinite loops in cyclic structures (like DAGs).

The central method `GenerateFormulaRecursive(Node node)` is responsible for building a formula for each node. If the node is a *leaf*, its formula is trivial and simply expresses that the corresponding variable equals 1 (i.e., it is active). If the node is a *composite gate*, such as an AND or OR gate, the method recursively generates formulas for all its children and combines them using the appropriate logical operator (`&` for AND, `|` for OR). So, for example for OrGate1 from Fig. 4.5 will have the following formula **OrGate1_id_triggered = (a1 | a2);** The result is stored as a named formula (with a suffix `_triggered`) which can be referenced elsewhere in the model.

A particularly interesting case is when the node has a `Cm` (countermeasure) gate type. This type represents a node that wraps another action with one or more countermeasures. The formula builder treats this as a special case. The first child of the Cm node is considered the *wrapped action*, and the remaining children are treated as *countermeasures*. The logic of the formula in this case expresses that the wrapped action is only triggered if all countermeasures fail (i.e., their effective status is false). This is encoded by recursively generating formulas for both the wrapped action and the countermeasures, and then combining them into a new formula using the conjunction of the wrapped action and the negation of all countermeasures' effective formulas. To illustrate how the formula looks like on particula example let's have a reference on Fig. 4.5 and have a look on countermeasure-

ment node (it is the only one). Its formula will look like **cm_orGate1_id_triggered = (orGate1_triggered & !d1_id_effective)**, where **!d1_id_effective = (d1_id=1)**, meaning that the countermeasurement node will be triggered in case if the attacker will exploit the OrGate1 and the defender won't activate his defense (d1=0).

For each formula generated, a unique name is created using a combination of the node's label and ID to ensure there are no collisions. These named formulas are stored in the `_formulas` dictionary and eventually returned as the final output of the `FormulaTreeBuilder`. The `Build()` method ensures that all root nodes are processed and that the resulting formulas form a tree-structured logical representation of the original ADT.

Finally, when the `FormulasSectionGenerator` is invoked via its `Generate` method, it simply collects and returns all the generated formula strings from the `FormulaTreeBuilder` and inserts them into the appropriate place in the PRISM model code.

This process not only enables a clean and modular PRISM model but also ensures logical consistency, efficient reuse, and a faithful representation of the semantics of countermeasures and composite attack-defense logic in the model.

Player Module Generation

The generation of player modules (attacker and defender) is implemented in the PlayerModuleSectionGenerator class, which implements the `ICodeSectionGenerator` interface. The `Generate` method receives a collection of nodes and flattens the tree structure using `TreeWalker.Flatten` to retrieve all nodes. It then separates them into attacker nodes (`IsAttackerNode`) and defender nodes (`IsDefenderNode`), and proceeds to generate the attacker and defender modules individually, followed by the game manager module.

For each player, a separate module is created (`module Attacker`, `module Defender`). Within each module, the `GenerateActions` method is called to create the transitions based on the available nodes. A transition includes checking whether it is the current player's turn, whether the selected node has not yet been activated (its associated variable equals 0), and whether the player has sufficient budget. If all conditions are met, the transition updates the variables: the node becomes activated (variable is set to 1), and the corresponding amount is subtracted from the budget. Additionally, at the end of each player's turn, a dedicated transition is added to end the turn. This transition advances the turn to the next player and sets a specific end-of-turn variable to `true`.

The game manager module is generated in the `Player Module Generator` class. It contains a single transition that is triggered when the turn variable equals 2. This transition assigns the value of the last formula from `FormulaTreeBuilder.Instance.FormulaNames` to the `top_event` variable and sets the `Turn` variable to 3, indicating that the game has reached its final state. This module ensures the connection between the players' actions and the global game conclusion condition.

Rewards and Label Section Generation

The `Rewards` block is generated through a straightforward iteration over the flattened list of nodes. For each node marked as `IsActionNode`, a corresponding reward entry is created. These entries assign predefined reward values to either the attacker or defender depending on the node type and its context. The reward entries reflect the utility or cost of specific actions taken by the players during the game.

The `Label` section is a pre-rendered and static part of the model that remains unchanged regardless of the input or global variables. It is included to facilitate state monitoring and control over the model's execution. Labels help in identifying certain game

conditions or phases and are essential for model checking and validation purposes.

4.3 Transformation results

Having outlined the internal architecture and detailed the logic behind the PRISM code generation process, it is now instructive to present a concrete example of how an Attack-Defense Tree is transformed into a formal model. The ADT from Fig 4.5 will be used as an input model. This section illustrates the end-to-end transformation pipeline by showing the resulting model structure for a small but representative ADT.

The ADT consists of three basic attacker actions—`a1`, `a2`, and `a4`—and one basic defender action—`d1`. In the structure of the tree, nodes `a1` and `a2` are combined using an OR gate (`OrGate1`); nodes `a2` and `a4` form a second OR gate (`OrGate2`). The result of `OrGate1` is wrapped in a countermeasure by `d1`. Finally, both `cm_OrGate1` and `OrGate2` are combined through an AND gate to define the top-level event of interest, referred to as `top_event`. All leaf nodes are annotated with cost attributes.

This logical structure is reflected in the generated PRISM model 4.1, which consists of multiple sections created by different generators as part of the transformation pipeline.

```
1  smg
2
3  // Define players
4  player attacker
5      Attacker, [attack_a1n2], [attack_a2n3], [attack_a4n7], [attack_end_turn
          ]
6  endplayer
7
8  player defender
9      Defender, [defense_d1n5], [defense_end_turn]
10 endplayer
11
12 player gameManager
13     GameManager, [check_gates]
14 endplayer
15
16
17 //Global variables
18 global turn : [0..3] init 0;
19 global top_event : bool init false;
20 //Attacker variables
21 global a1_n2: [0..1] init 0;
22 global a2_n3: [0..1] init 0;
23 global a4_n7: [0..1] init 0;
24 //Defender variables
25 global d1_n5: [0..1] init 0;
26
27 global attacker_done : bool init false;
28 global defender_done : bool init false;
29
30
31
32 const int INIT_ATTACKER_BUDGET;
33 global attacker_budget: [0..65] init INIT_ATTACKER_BUDGET;
34 const int INIT_DEFENDER_BUDGET;
35 global defender_budget: [0..10] init INIT_DEFENDER_BUDGET;
36
37
38 formula OrGate1_n1_triggered = (a1_n2=1 | a2_n3=1); //<----OrGate1
```

```
39 formula n5_effective = (d1_n5=1);
40 formula cm_OrGate1_n4_triggered = (OrGate1_n1_triggered & !n5_effective);
       //<----cm_OrGate1
41 formula OrGate2_n6_triggered = (a2_n3=1 | a4_n7=1); //<----OrGate2
42 formula top_event_n0_triggered = (cm_OrGate1_n4_triggered &
       OrGate2_n6_triggered); //<----top_event
43
44 module Attacker
45     [attack_a1n2] turn=1 & a1_n2=0 & attacker_budget>=15 -> (a1_n2'=1) & (
           attacker_budget'=attacker_budget-15);
46     [attack_a2n3] turn=1 & a2_n3=0 & attacker_budget>=30 -> (a2_n3'=1) & (
           attacker_budget'=attacker_budget-30);
47     [attack_a4n7] turn=1 & a4_n7=0 & attacker_budget>=20 -> (a4_n7'=1) & (
           attacker_budget'=attacker_budget-20);
48     [attack_end_turn] turn=1 -> (turn'=2) & (attacker_done'=true);
49 endmodule
50 module Defender
51     [defense_d1n5] turn=0 & d1_n5=0 & defender_budget>=10 -> (d1_n5'=1) & (
           defender_budget'=defender_budget-10);
52     [defense_end_turn] turn=0 -> (turn'=1) & (defender_done'=true);
53 endmodule
54 module GameManager
55     [check_gates] turn=2 -> (top_event'=top_event_n0_triggered) & (turn'=3)
           ;
56 endmodule
57
58
59 rewards "attacker_cost"
60     [attack_a1n2] true : 15;
61     [attack_a2n3] true : 30;
62     [attack_a4n7] true : 20;
63     [attack_end_turn] true : 0;
64 endrewards
65 rewards "defender_cost"
66     [defense_d1n5] true : 10;
67     [defense_end_turn] true : 0;
68 endrewards
69
70
71 label "top_event_reached" = top_event;
72 label "system_secure" = !top_event;
73 label "end_game" = (turn = 3);
```

Listing 4.1: PRISM code for ADT on Fig 4.5

The **Players Block**, defined at the beginning of the model (line 3), declares three players: `attacker`, `defender`, and `gameManager`. Each is associated with their corresponding modules and the action labels they control.

The **Global Variable Block** introduces shared variables (line 17) for turn control (`turn`), the final evaluation outcome (`top_event`), and one variable per basic action node. For example, `a1_n2`, `a2_n3`, and `a4_n7` are boolean variables representing whether attacker actions `a1`, `a2`, and `a4` have been executed. Likewise, `d1_n5` represents the defender's countermeasure. Each variable name incorporates both the label and unique identifier of the node to avoid conflicts. The model also defines attacker and defender budgets and their respective initialization constants.

The **Formula Block**, generated by the `FormulasSectionGenerator`, encodes logical relationships between composite nodes. The OR gates are translated into Boolean formulas, such as on line 38. The countermeasure node is processed by the adapter and formula

builder to ensure that it suppresses `OrGate1` when active (line 40). The root AND gate is encoded on line 42.

The **Modules Block** defines the behavior of each player (line 44). For the `Attacker` module, each basic action is represented by a guarded command that checks if it is the attacker's turn, verifies the availability of budget, and updates the corresponding variable upon execution. A similar pattern is followed in the `Defender` module (line 50), where the defender can activate `d1` if sufficient budget is available. The `GameManager` module (starts from line 54) includes a transition that assigns the final formula evaluation (`top_event_n0_triggered`) to the variable `top_event` once both players have completed their turns. Reward structures (line 59-attacker, line 64-defender) are generated for both players. These are straightforward mappings from each action to its associated cost.

Finally, the **Label Section** (line 71) provides static labels for use in verification. These include whether the `top_event` was reached, whether the system is secure (i.e., `top_event` did not occur), and whether the game has reached its end state.

The generated PRISM code corresponds precisely to the structure of the ADT shown in Figure 4.5. All major code blocks are present: global variables, formulas, modules, rewards, and labels. Each transition in the attacker and defender modules directly reflects the nodes and edges of the original tree structure. Furthermore, all cost values defined in the ADT are correctly transferred into the corresponding reward structures. This example confirms that the transformation pipeline faithfully preserves both the logic and quantitative attributes of the input tree.

4.4 Queries

To illustrate mentioned above advantages in a realistic setting, it is included as a case study a DAG-structured ADT on Fig 4.5. The generated code for it you can see in Listing 4.1.

It is possible to validate the model through different kinds of properties. The first property that will be checked is:

```
R{"attacker_cost"}min=? [F "top_event_reached" & "end_game"]
```

This property computes the minimum cost that the attacker needs to spend to reach the top-level event. It is important to note that the model includes functionality allowing the defender to skip turns, which means the model naturally considers scenarios where the defender does not spend any resources. Also, this property is evaluated assuming unlimited budgets for both the attacker and the defender, thus exploring all possible attack and defense strategies without budget constraints. The result is 30, which means that attacker can activate **a2** node, exploiting both **OrGate1** and **OrGate2**, leading to the top event.

The next property that will be checked is:

```
R{"attacker_cost"}min=? [F OrGate1_n1_triggered ]
```

This property also calculates the minimum attacker cost, as the previous one, but it differs in the way that it checks what is the minimum attacker cost to exploit the **OrGate1**. The result is 15, which is obvious, because the attacker just need to activate the **a1** node.

Now lets explore the costs from defenders perspective:

```
R{"defender_cost"}min=? [F "system_secure" & attacker_budget=0 & "end_game"]
```

This property checks what is the minimal defender cost to keep the system secure in the end, assuming that the attacker has wasted entire budget (force attacker to make moves). It is checked with unlimited budgets. The result is 10, because the **top event** is the `AND gate`, which means that both **OrGate1** and **OrGate2** should be activated, and the **d1** node prevents the attack from the left side of the tree - via **ORGate1**.

Finally, it is possible to check probabilistic properties:

```
Pmin=? [ F "system_secure" & "end_game"]
```

This property validates the following question: what is the minimum probability that top event will not occur (i.e. the system will always be secure)? This property was verified with budget constraints: attacker's budget is 20, defender's budget is 10 (or unlimited). The result is 1, which means, that given attacker's and defender's budget constraints, the system will be secure always.

Despite the fact that the ADT model can be analyzed using a variety of formal properties, the core interest in using ADT lies in the ability to explore the trade-offs between different objectives. In particular, calculating the Pareto Front provides valuable insights into how these trade-offs manifest in the system. The following chapter focuses on the computation and interpretation of the Pareto Front in the context of the ADT model.

4.5 Summary

This chapter discussed the architecture and implementation of ADTransformer, a novel tool that automates the transformation of Attack-Defense Trees into formal models suitable for verification in PRISM. The architecture follows a modular design, with each component responsible for a specific task in the transformation workflow, from parsing ADTool XML inputs to generating PRISM, compatible code and conducting formal analysis. This automation bridges the gap between intuitive ADT modeling and rigorous model checking. It was also demonstrated how the generated model and queries can be used to extract valuable and relevant information from the ADT. This enables answering both high-level questions about the system as a whole and more specific ones related to particular gates and system states.

Chapter 5

Pareto Front Analysis

One of the most useful and instructive results that can be obtained from an Attack-Defense Tree study is the Pareto Front. It enables analysts to comprehend how boosting one side's resources affects the necessary investment from the other side by capturing the essential trade-offs between attacker and defense methods. To put it another way, the Pareto Front shows that minimal defender effort is required to stop attacks of different strengths; this viewpoint directly influences choices about risk management and resource allocation.

The PRISM model checker has one significant limitation when using the SMG (Stochastic Multi-Player Game) module type: the tool does not support multi-objective queries. As a result, it is not possible to directly query multiple metrics simultaneously. To overcome this, the analysis instead performs a systematic single-objective evaluation for each fixed combination of attacker and defender budgets. This allows the tool to extract partial but meaningful information about the Pareto front — specifically, whether a given budget allocation is sufficient for the attacker to succeed or not. Although it does not produce the full multi-dimensional frontier, this method efficiently identifies Pareto-optimal boundaries by determining the minimal defender investment required to block each possible attacker budget.

5.1 Property-Guided Pareto Analysis

As it was mentioned in Section Pareto Front, in the context of Attack-Defense Trees, we are interested in identifying the minimal amount of defender resources required to successfully block an attacker with a given budget. Rather than computing the entire multi-objective Pareto frontier explicitly, the following PRISM property is used to extract partial information about it:

```
<<attacker>> Pmax=? [ F "top_event_reached" & ("end_game" | "deadlock") ]
```

This property answers the question: Is it possible for the attacker to reach the top-level goal? Because of the fact that this property runs with varying attacker and defender budgets, the question can be interpreted as: Is it possible for the attacker to reach the top-level goal under the current attacker and defender budget constraints? If the result is 1 (true), the attacker can succeed despite the defense — meaning the defender's strategy is insufficient. If the result is 0 (false), the system is considered secure under the given budget allocation.

This property provides binary feedback about whether a point lies above or below the defender's effective threshold for a given attacker budget. As such, it only gives partial

information about the Pareto front: it does not explicitly compute all non-dominated trade-off points across the entire strategy space, but instead identifies Pareto-optimal defender budgets for fixed attacker budgets. Each such point indicates a boundary at which the defender's investment becomes sufficient to block the attack, thus contributing to the frontier of non-dominated solutions. Therefore, the property serves as a Pareto classifier, because it tells whether a defender strategy is dominated or potentially Pareto-optimal (i.e., attacker is stopped at minimum cost). It enables efficient construction of the front by exploring only the minimal points at which the transition from attacker success to failure occurs.

5.2 How ADTransformer collects and processes the information

To explore the entire attacker-defender strategy space, ADTransformer systematically iterates over all possible combinations of attacker and defender budgets:

```
for(int attBudget = 0; attBudget <= MAX_ATT; attBudget++) {
    for(int defBudget = 0; defBudget <= MAX_DEF; defBudget++) {
        // run PRISM with given attacker and defender budgets
    }
}
```

LISTING 5.1: Pseudocode of how budgets are iterated

For each pair (attBudget, defBudget), the tool sets up the PRISM model with those values and queries the above property. The result of each query determines whether the attacker can succeed. To find Pareto-optimal defense budgets, the tool applies the following logic:

- For each fixed attacker budget, iterate defender budgets in increasing order.
- Stop at the first defender budget where the PRISM property returns 0. This point (attBudget, defBudget) is:
  - The minimal defender investment to prevent the attacker from reaching the goal.
  - Pareto-optimal, because lowering the defender budget any further would allow the attacker to win.

This guarantees that each point in the Pareto front reflects a trade-off: increasing attacker capability must be met with greater defender investment. Conversely, excessive defense is avoided unless required. This method ensures the Pareto condition:

- No other (defender budget, attacker budget) pair is strictly better in both dimensions (less defense, more attacker success).
- The outer loop ensures the attacker budget is fixed per iteration.
- The inner loop ensures minimization of defender cost for each attacker capability.

5.2.1 On the Limitations of Alternative Properties

An alternative idea that was considered during the development of the method was to compute the Pareto-optimal points using a reward-based property, that was showed in Section 4.4:

```
R{"attacker_cost"}min=? [ F "top_event_reached" ]
```

This property queries the minimal attacker cost required to achieve the goal under the current defender budget. The initial intuition was that, by systematically increasing the defender budget and observing the corresponding minimal attacker cost, it would be possible to infer the Pareto front by identifying points where increasing the defender budget forces an increase in attacker cost.

However, this approach proved to be unreliable in practice for the following reason. The "min" reward property assumes optimal play only from the attacker's perspective, and does not guarantee that the defender is using a strategy consistent with the broader trade-off space (e.g., with prior defender investments at lower budgets). In other words, the property may produce results where the defender chooses a suboptimal strategy that accidentally enables a cheaper attack path not aligned with the actual defense trends. As a result, the computed attacker cost can appear to decrease even as defender investment increases — which is counterintuitive and incorrect for Pareto front analysis.

This behavior stems from the fact that the reward property optimizes only for the attacker's perspective in a single query, without enforcing consistency across the defender's strategy space. Therefore, the method was abandoned in favor of the double iteration approach (over both attacker and defender budgets), which explicitly evaluates whether the defender can prevent the attack under each possible budget allocation. This more exhaustive method ensures that the resulting Pareto front reflects consistent and meaningful trade-offs between attacker and defender investments.

5.2.2 Optimizations in Pareto Front Computation

Thus, the tool constructs the Pareto front as the set of (attackerBudget, defenderBudget) pairs where The attacker fails (PRISM value is 0), and any lesser defender budget would lead to attack success (value becomes 1 again).

It is worth noting that during the process of generating the Pareto Front, various optimization methods for finding Pareto-optimal points were applied. First of all, the iteration over attacker and defender budgets does not proceed by simply iterating from 0 to the maximum attacker/defender budget, as shown on Listing 5.1, but rather by iterating over all possible combinations of attacker/defender cost sums (Listing 5.2). This approach guarantees that the number of iteration steps is reduced.

```
1 List<Integer> attackerBudgets = getAttackerBudgetList(); // e.g., [0, 10,
     20, 30, 40, ...]
2 List<Integer> defenderBudgets = getDefenderBudgetList(); // e.g., [0, 5,
     15, 25, 35, ...]
3
4 for (int attBudget : attackerBudgets) {
5     for (int defBudget : defenderBudgets) {
6         // run PRISM with given attacker and defender budgets
7     }
8 }
```

Listing 5.2: Pseudocode of how budgets are iterated withing different cost sums

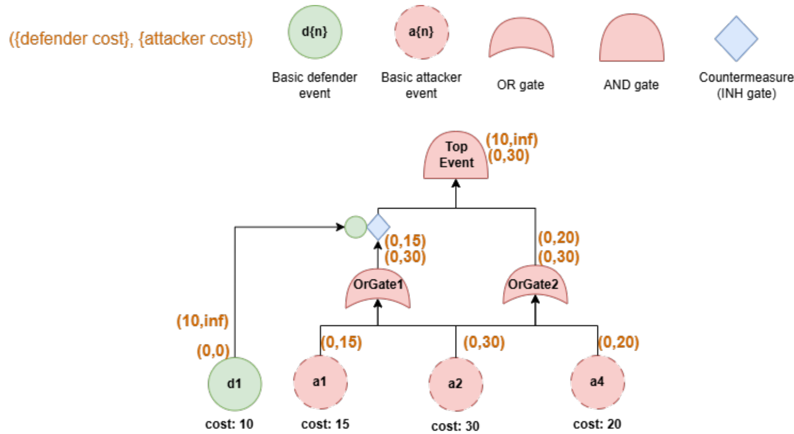

FIGURE 5.1: Calculated Pareto Front for ADT from Fig. 4.5

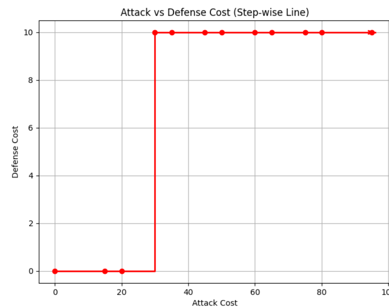

FIGURE 5.2: Pareto Front plot for 5.1

Additionally, to optimize the Pareto front computation, an early termination technique was applied: the iteration over defender budgets is stopped when the property returns 1 for all defender budgets. This indicates that, for the current attacker budget, no defender budget is sufficient to protect the system — meaning this is the last point that will appear on the Pareto front for this attacker budget.

5.2.3 Case study

In Chapter 4.4, an example was provided to illustrate how the generated model and specific properties can be used to extract useful information from an ADT. Here, an example based on the same ADT (Fig. 4.5) will be presented to demonstrate how the tool computes the Pareto Front.

In this DAG-structured ADT 5.1, there are two OR gates — OrGate1 and OrGate2 — combined into the top event. d1 is a countermeasure for OrGate1, meaning that if d1 is active, the attacker cannot bypass this countermeasure to reach the top event. At the same time, a2 is a shared basic attacker event across both OR gates. It is more expensive than a1 or a2, and by applying the standard formula for computing the cost of OR gates, one would obtain a cost of 15 for OrGate1 and 20 for OrGate2. This would imply that the cost of the top event is 35. However, since this is a DAG-structured ADT, the attacker can activate a2 to compromise both OrGate1 and OrGate2 simultaneously, resulting in a top event cost of 30 — which is lower than 35. ADTransformer is able to detect such cases and compute the minimal attack cost accordingly. The output of ADTransformer can be

seen in Fig. 5.2.

5.3 Summary

In this chapter it was demonstrated how ADTransformer enables property-guided Pareto front analysis, providing users with insight into the trade-offs between attacker and defender strategies. Unlike traditional approaches, ADTransformer supports complex query-driven analysis and automatically computes multi-objective trade-offs using PRISM's verification engine. We discussed how the tool handles limitations in property selection, optimizes performance for large models, and visualizes the resulting Pareto fronts.

Chapter 6

Experimental Evaluation

In the previous chapters, the structure of the tool, the process of generating the PRISM model, and the method for computing the Pareto Front were described. In this chapter, performance measurements will be conducted: the model generation speed and the Pareto Front generation speed will be evaluated.

6.1 Case study

To demonstrate the capabilities of proposed approach, lets consider a representative ADT scenario derived from a classical example presented by Kordy et al. [27]. The scenario models an attacker's objective to gain access to a victim's bank account, either via an ATM or online banking, while capturing various countermeasures and associated costs.

Figure 6.1 illustrates the DAG-structured ADT used in this case study. The top-level attack goal (*access bank account*) can be achieved through one of two disjoint subgoals: compromising the ATM channel or exploiting online banking credentials. Each path is further decomposed into basic attack steps (e.g., *learn PIN*, *get password*, *get username*) and corresponding defensive actions (e.g., *cover keypad*, *strong password*, *SMS authentication*) taken by the defender.

The ADT includes numerical cost values assigned to the leaf nodes, representing the estimated effort or monetary cost of performing or defending against the corresponding action. These values are used to support quantitative evaluation through both single-metric queries and multi-objective trade-off analysis.

The subsequent subsections present two perspectives on analyzing this ADT using PRISM-based transformation. Section 6.1.1 explores how temporal logic properties can be used to extract specific insights from the model and how, based on them, make potential decisions. Section 6.1.2 demonstrates how Pareto-optimal strategy profiles can be computed.

6.1.1 Queries

This section will illustrate how to capture valuable insights from ADT using PRISM code (see Listing 8.2) and make decisions, based on the queries (or PRISm properties).

Lets assume that we are the system owner and we want to protect the system by finding the weakest spot in the system. As can be seen from the ADT in 6.1, the system consists of two disjoint subtrees: Online banking and ATM. These represent the two main paths through which an attacker can compromise the system—either via online banking or via

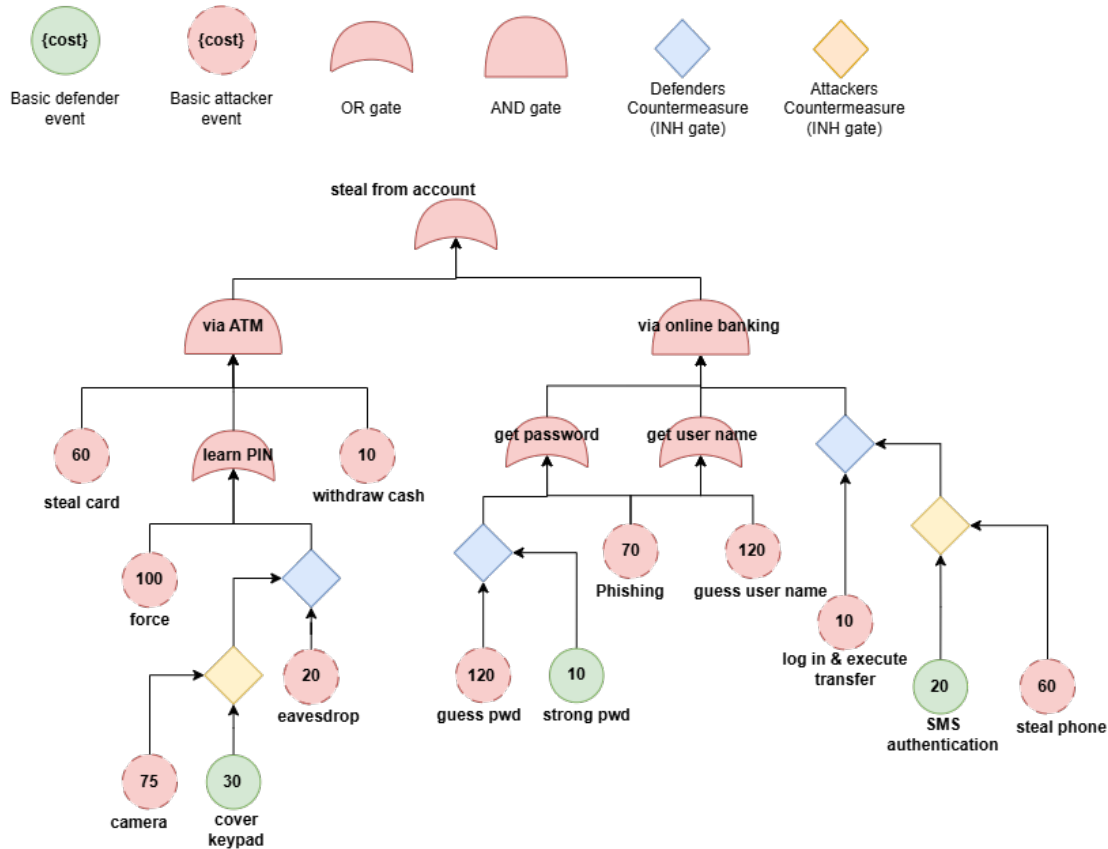

FIGURE 6.1: Attack-defense tree representing a money theft scenario, adapted from [27]

ATM; compromising just one of them is sufficient. To begin, we will analyze which of the subtrees is cheaper to compromise. For this purpose, we will use the following properties:

```
R{"attacker_cost"}min=? [ F viaOnlineBanking_n11_triggered ]
R{"attacker_cost"}min=? [ F ATM_n1_triggered ]
```

These properties compute the minimum cost of an attack through each of the subtrees, assuming that the defender takes no action. The properties are evaluated under the assumption that the attacker has unlimited resources. The results are as follows:

- **via online banking**: 80
- **via ATM**: 90

Now, we evaluate the same properties under the assumption that the defender has taken all possible measures to protect each of the subtrees. The evaluation is again performed assuming that both the defender and the attacker have unlimited resources.

```
R{"attacker_cost"}min=? [ F viaOnlineBanking_n11_triggered&defender_budget=0 ]
R{"attacker_cost"}min=? [ F ATM_n1_triggered&defender_budget=0 ]
```

The results are as follows:

- **via online banking**: 140
- **via ATM**: 165

Based on these results, we can conclude that the preferred target for the attacker, in terms of minimizing the cost of the attack, is the path through online banking. This is because it is cheaper to compromise than the ATM path, both under the assumption that the defender does nothing and under the assumption that the defender takes all possible protective measures.

Now, let us dive a bit deeper. As shown in Fig. 6.1, the online banking subtree consists of the nodes *get password*, *get username*, and *log in & execute transfer*. We now proceed to evaluate the following property:

```
R{"attacker_cost"}min=? [ F getPassword_n12_triggered&getUserName_n17_triggered ]
```

This property computes the minimum cost of attacking the *get username* and *get password nodes*. The result is 70. From the result and the system diagram (Fig. 6.1), it is evident that compromising the Phishing node is sufficient for the attacker to obtain both the password and the username.

Now, let us evaluate the likelihood of a successful attack via the online banking path in the case where Phishing is not activated by the attacker (e.g., due to countermeasures that effectively block the Phishing node):

```
Pmax=? [F viaOnlineBanking_n11_triggered&Phishing_n16=0&attacker_budget=0
&defender_budget=0]
```

The result is 0. This means that the attacker has no available paths to compromise the system if Phishing is prevented, assuming the defender has taken all possible protective measures. This indicates that implementing anti-phishing mechanisms is sufficient to fully protect the Online banking path.

Next, we compute the minimum cost of defending the Online banking subtree under the assumption that Phishing is not activated. In this case, the property should specify the attacker's maximum available budget, reduced by the cost of attacking the Phishing node, to ensure the results are accurate.

```
R{"defender_cost"}min=? [F Phishing_n16=0&!viaOnlineBanking_n11_triggered
&attacker_budget=0]
```

The result is 10. This means that, under the assumption that Phishing is not activated, the defender only needs to invest a cost of 10. As shown in Fig. 6.1, this cost corresponds to the strong password node. In other words, by deploying anti-phishing mechanisms, it is sufficient for the defender to activate the strong password node along with those anti-phishing mechanisms in order to secure the Online banking path.

The total defense cost, after applying these countermeasures, can be calculated as the combined cost of the strong password node and the anti-phishing mechanisms. Importantly, this strategy ensures that the system remains protected even in scenarios where, for instance, a phone is stolen and the SMS authentication mechanism is not activated.

6.1.2 Pareto front

To illustrate how our Pareto front analysis works in practice, we apply it to the complex DAG-structured Attack-Defense Tree introduced in Section 6.1.1 on Fig. 6.1. This model simulates a realistic cybercrime scenario in which an attacker aims to steal funds from a bank account. The tree includes multiple attack paths — such as phishing, credential theft, and ATM withdrawal — along with a range of countermeasures like SMS authentication, strong passwords, and keypad protection. The similar case was introduced in [11] in section *Synthetic ADTs*, Fig. 7.

Due to its size, depth, and shared substructures, this tree is well-suited for evaluating the expressiveness and scalability of our tool. It also reflects real-world complexities such as conditional defenses and overlapping strategies, making it an ideal candidate for testing budget-sensitive trade-offs between attacker and defender.

The figure 6.2 below visualizes the **Pareto front** computed from this tree. Each point on the curve represents a *Pareto-optimal pair of attacker and defender budgets.* For a fixed attacker budget A, the corresponding defender budget D is the *minimum investment* required to prevent the attacker from reaching the top-level goal (i.e., the "top_event"). These points were obtained by iterating through all combinations of attacker and defender budgets and evaluating the PRISM property described in Section 5.1:

```
<<attacker>> Pmax=? [ F "top_event_reached" & ("end_game" | "deadlock") ]
```

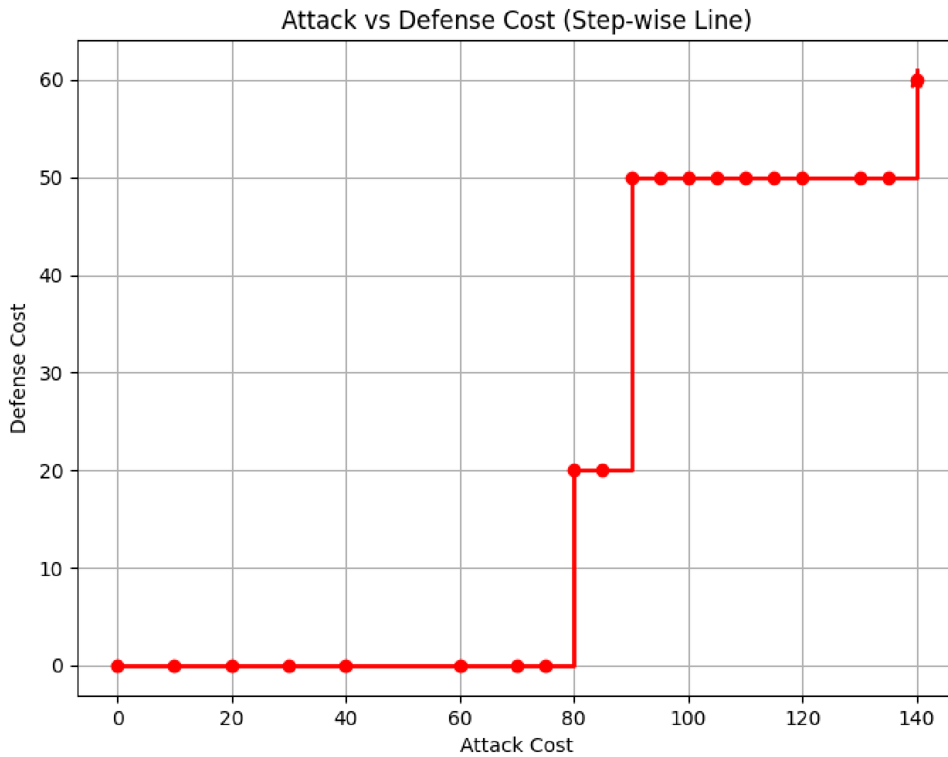

FIGURE 6.2: Pareto Front plot for 6.1

Whenever this property returned 0, it indicated that the attacker could no longer reach their goal under the current budget conditions, making the defender budget Pareto-optimal for that attacker level.

The computed Pareto front was also cross-validated against the results obtained with the Binary Decision Diagram (BDD)-based algorithm presented in [11] (see Fig. 8 therein). The ADTransformer successfully reproduced the same Pareto-optimal points as the BDD-based method, confirming the correctness and precision of its computations. While the computation speed of our approach is not yet as fast as highly optimized BDD-based solvers, it provides an accurate and scalable alternative that integrates seamlessly with the PRISM model-checking workflow.

A notable feature in the graph is the vertical line starting at the point $(140, 50)$. This represents the threshold beyond which *no defender strategy is effective*, regardless of the investment. Once the attacker reaches a budget of 140 or more, the defender can no longer prevent a successful attack, even with unlimited resources. In Pareto terms, this corresponds to a point like $(140, \infty)$, and the vertical line highlights the transition from feasible to infeasible defense.

This boundary effectively partitions the state space into "secure" and "insecure" zones. Overall, this case study demonstrates how proposed tool can efficiently compute and visualize attacker-defender trade-offs. The result is a practical decision-support mechanism for cybersecurity professionals working on budget allocation and strategic defense planning.

6.2 Test Environment

All experiments in this chapter were conducted on a machine with the following hardware configuration:

- AMD Ryzen 5 5600H with Radeon Graphics, 3.3 GHz, 6 physical cores, 12 logical processors
- RAM: 16 GB
- Operating System: Windows 10

The model checking was performed using PRISM-games 3.2.1, a specialized version of the PRISM model checker supporting SMG. The models were generated automatically using the ADTransformer tool developed in this work. The ADTransformer tool is implemented in ASP.NET 9 and executed as a standalone console application.

6.3 Model Generation

In section 4.2.3, it was shown how the ADTransformer generates models, and a trivial case of validating the generated model using PRISM properties was discussed. The goal of this section is to demonstrate how quickly the ADTransformer is able to generate models. Additionally, a more complex ADT and an example of properties that can be used to extract specific information will also be discussed.

6.3.1 Test setup

The timing measurements were performed on randomly generated ADTs. This approach uses many randomly generated ADTs (DAG and Tree structured) to statistically compare the tool's performance. After setting a maximum number of children n, nodes with random properties (gate type, attack/defense type, number of children, costs) are recursively generated until the tree contains $|N| = n$ nodes. Trees of sizes 2, 50, 100, 200, and 400 were generated. For each tree size, 10 to 15 examples were used to calculate the average computation time. Here, the tree size refers to the total number of basic attacker and defender events combined and does not take into account the intermediate nodes, such as AND, OR and INH gates.

FIGURE 6.3: Model generation plot

6.3.2 Model Generation Time

The time measurement includes the evaluation of three separate modules of the tool that process the input XML file describing the ADT: the Parser, the Adapter, and the PRISM Code Generator. These stages are described in section 4.1 and illustrated in Fig. 4.2. Figure 6.3 shows the dependence of PRISM model generation time on the size of the ADT (treeSize). The experiments were conducted on trees of size 2, 50, 100, 200, and 400. For each tree size, 10–15 randomly generated examples were used to compute the average generation time (as described in the previous subsection 6.3.1).

Results show that while model generation time rises with tree size, the overall increase is very slight. The generation time curve tends to flatten after initially increasing up to tree size 100, with very slight differences between sizes 100, 200, and 400. The creation time is still less than 5 milliseconds, even for the biggest tested trees (400 nodes).

These outcomes show how effective the ADTransformer tool is and how it can manage big ADTs with little effect on performance.

6.3.3 Lines of Code Analysis

In addition to measuring model generation time, we also evaluated how the size of the Abstract Data Tree (ADT) affects the size of the generated PRISM model in terms of lines of code (LOC). This metric helps assess how the complexity of the generated model scales with the input tree structure.

Figure 6.4 presents the relationship between tree size and the number of lines of code in the corresponding PRISM model. The plot uses a logarithmic scale on the Y-axis to better visualize the rate of growth. The experiments were performed on trees of size 2 up to 400. For each size, between 15 to 20 randomly generated tree structures were used to calculate the average number of LOC in the generated models.

The results show that LOC grows rapidly for small tree sizes (up to around 100 nodes), after which the growth slows and gradually flattens. This indicates a sublinear or possibly logarithmic growth trend for larger trees. The plateau in LOC growth suggests that the PRISM model structure becomes more regular and saturated as tree depth increases.

Overall, this behavior confirms that the ADTransformer tool produces scalable and concise PRISM models, maintaining manageable code size even for large ADTs.

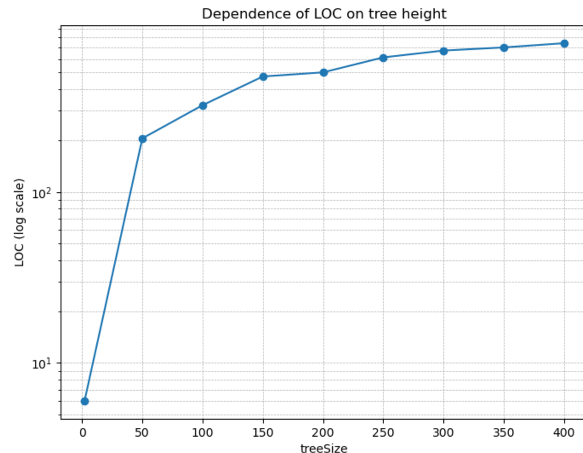

FIGURE 6.4: Dependence of LOC on tree size (logarithmic scale)

6.4 Pareto Front computation

In Chapter 5, the process of computing the Pareto front using the ADTransformer tool was discussed in detail. In this section, we will evaluate the computation time. The measurement focuses exclusively on the execution time of the Prism Runner module, described in Section 4.1. This module is responsible for repeatedly executing the property discussed in Section 5.2 and for collecting and processing the corresponding validation results.

6.4.1 Test setup

The timing measurements for Pareto front computation were conducted using randomly generated ADTs. A set of randomized ADTs was generated using a custom Python script. The generator creates valid tree-structured ADTs in XML format by recursively assigning randomly chosen node types (AND/OR gates, attacker or defender actions) and connecting them into a tree. Each basic action is annotated with a random cost. The size of the tree and value ranges were configurable, allowing the evaluation of tool scalability under controlled synthetic scenarios. Following the same methodology as described in the model generation evaluation (see Section 6.3.1). These ADTs include both tree-structured and DAG-structured instances. For this experiment, the test set consisted of ADTs with sizes ranging from 2 to 18 nodes, with 10 to 15 randomly generated trees per size. Each ADT was constructed recursively by setting a maximum number of children per node and randomly assigning properties such as gate type, attack/defense labels, child count, and costs. The resulting set was used to evaluate the average computation time of the Pareto front analysis for small to moderately sized ADTs. Here, the tree size refers to the total number of basic attacker and defender events combined and does not take into account the intermediate nodes, such as AND, OR and INH gates.

6.4.2 Runtime computation

Fig. 6.5 shows dependence of execution time on tree size. The results demonstrate that computational performance scales significantly with tree size. The runtime increases from approximately 2 seconds for small trees with size of 2 to over 2000 seconds for larger

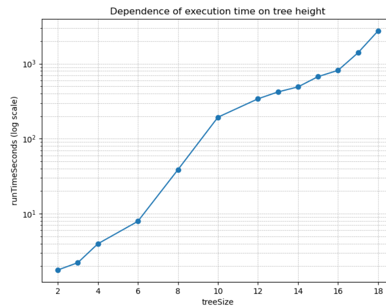

FIGURE 6.5: Run time evaluation for Pareto Front computation

trees with size of 18, exhibiting what appears to be exponential growth with considerable fluctuations.

The results indicate relatively slow performance characteristics. When compared to the Naive algorithm described in the [10] on Fig 8.1 (Summary of all the pairwise comparisons, where 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), the computational time is remarkably close to that baseline approach. This suggests that while the algorithm maintains competitive timing relative to naive implementations, there is substantial room for performance optimization.

All experiments in this chapter were conducted on a machine with the hardware configuration represented in 6.2 Test Environment. The experiments conducted in Copae's work used an Intel Core i5-12600K (3.7 GHz) and 16 GB RAM, running Python 3.12 and the Gurobi solver. While their setup is not identical, the computational capabilities of both machines are relatively similar in terms of core count and memory, although the i5-12600K may have a slight advantage in single-thread performance.

Unfortunately, I was unable to run all the experiments from [10] on my local setup, as the full codebase and test cases were not publicly available on GitHub at the time of writing. This limitation prevents a fully objective, side-by-side performance comparison between my algorithm and the ones described in that work using identical test conditions.

Nevertheless, given the similarity in hardware specifications, I make a reasonable assumption that the performance characteristics would not differ drastically if both implementations were tested under the same environment. Therefore, I base my performance comparisons on the empirical data and figures reported in their publication.

Despite the computational overhead, ADtransformer demonstrates high accuracy in computing the Pareto front. The algorithm's precision in identifying optimal trade-offs compensates for its slower execution speed, making it a viable approach when solution quality is prioritized over runtime efficiency. A detailed analysis of the Pareto front accuracy will be presented in section 6.1.2.

Chapter 7

Discussion & Conclusion

In the previous chapters, the tool developed as part of this thesis — ADTransformer — was discussed in detail. Its architecture was presented, along with the process of generating a PRISM model based on input from ADTool, a tool for modeling attack–defense trees. Additionally, the process of computing the Pareto front was explained. Section 6.1 demonstrated how the tool can be used to analyze generated ADT models via PRISM properties and compute the corresponding Pareto front.

This chapter summarizes the advantages and limitations of the tool and presents ideas for its future extension.

7.1 Advantages of ADTransformer

Formalizing an ADT can be time-consuming, especially when the tree is large and complex. The description process requires significant human effort and time. ADTransformer facilitates rapid generation of trees based on a visual model, effectively automating the model formalization process. This is one of the key advantages of the developed tool.

In addition to tree generation, the tool formalizes the visual model by producing ready-to-use PRISM code that describes the ADT. This code enables in-depth analysis of the model, as demonstrated in the case study in Section 6.1.1. With the appropriate combination of PRISM properties, it becomes easy to identify system vulnerabilities and simulate what-if scenarios.

The generated PRISM code is designed in a way that allows the user to capture potential system states that may be difficult to detect manually. For example, it can answer questions such as: *"What is the minimal cost to attack a given node, assuming specific initial conditions?"* or *"What is the maximum probability of compromising a node, under certain assumptions?"*.

So far, as shown in Case study, we have mostly analyzed relatively simple properties over a small-scale model. However, the expressiveness of PRISM's property specification language enables users to investigate a wide range of possible system states using the generated model.

That said, alongside its clear advantages, the tool also has a number of limitations that must be acknowledged.

7.2 Limitations

One of the obvious limitations of the model is the speed of computing the Pareto front. As discussed in Chapter 5, the Pareto front is a highly attractive method for analyzing

ADTs. However, the method proposed in that section for computing the Pareto front did not demonstrate promising performance in terms of computation time.

When comparing this method to the algorithms discussed in Chapter 8 (Experiments) of [11], the proposed approach falls significantly behind the algorithm developed by Danut-Valentin Copae in terms of efficiency.

Another limitation of ADTransformer lies in the lack of full automation for checking PRISM properties. In the current setup, the user still needs to manually formulate relevant properties and validate them using PRISM games. This adds additional effort for the user and limits the usability of the tool for non-expert users unfamiliar with PRISM syntax and model checking workflows.

7.3 Future Work

The architecture of the tool, discussed in Section 4.1.Tool Architecture, adheres to all the principles of a Modular Architecture. This type of architecture is characterized by the fact that each component has minimal dependencies on others, modules interact through well-defined interfaces—making them easily replaceable—and new modules can be added without requiring significant changes to the existing codebase. Therefore, the potential for extending the tool's functionality is considerable.

One obvious improvement would be replacing the current console-based interface with a Graphical User Interface (GUI). This would significantly enhance the tool's usability and visual appeal, making user interaction with the tool's functionalities much more convenient.

Another potential improvement involves integrating a process for verifying custom properties defined by the user. Since the tool already embeds the PRISM-games engine, it would be feasible to implement a module that takes user-defined properties, automatically generates the appropriate PRISM specifications, runs validation through the integrated PRISM engine, and outputs the results. This enhancement would partially address one of the current limitations of ADTransformer mentioned above—its lack of full automation in property verification.

This thesis primarily investigates the transformation of Attack-Defense Trees (ADTs) into two-player game automata and encapsulates this approach within a unified framework. Nevertheless, several promising directions for future work are envisaged. One key extension involves the incorporation of probabilistic support, as current tool capabilities are limited to quantitative metrics such as cost. Given that ADTs can naturally support the computation of probabilistic scenarios, this represents a valuable addition [18]. Another important enhancement is the introduction of dynamic or time-dependent ADTs, allowing the framework to account for time-sensitive threats by associating timing constraints with attack and defense actions [1]. Furthermore, support for concurrency is crucial for accurately modeling real-world cybersecurity interactions, where multiple actions may occur simultaneously [21]. Finally, improving optimization capabilities is essential, as the current implementation shows weak performance in computing the Pareto front. This limitation could be addressed by integrating more efficient algorithms, such as the one proposed by Danut-Valentine Copae [10].

7.4 Summary

In this thesis, we presented a tool that enables in-depth analysis of Attack–Defense Trees. The main advantage of this tool is the formalization of ADTs based on their visual representation through the automatic generation of a PRISM model. The proposed method of

analysis through the transformation of ADTs into 2-player game automata is a significant step forward in the field of attacker–defender interaction analysis within cybersecurity. The combination of the structured representation of ADTs and the mathematical rigor of game automata helps overcome the main shortcoming of existing tools — their limited expressiveness.

Beyond its theoretical contribution, the ADTransformer has practical relevance. It allows security analysts to convert ADTs from ADTool into formally analyzable models without requiring deep knowledge of formal methods or manual model rewriting. Practitioners can evaluate attack scenarios not only in terms of static quantitative metrics, but also in terms of strategic trade-offs, multi-objective optimization, and equilibrium-based reasoning. This enables risk assessment processes to move from simple heuristic scoring towards rigorous, repeatable, and explainable decision support. Furthermore, the output models are compatible with PRISM's extensive analysis capabilities, offering integration potential in larger security analysis pipelines or continuous assessment workflows.

Chapter 8

Appendix A

Code Listings

```
public class Node
{
    public string Id { get; set; }
    public string Label { get; set; }
    public GateType? GateType { get; set; }
    public PlayerType? Owner { get; set; }
    public int? Cost { get; set; }
    public List<Node> Children { get; set; } = new();
    public bool IsLeaf => Children.Count == 0;
    public bool IsActionNode { get; set; } = false;

    public bool IsAttackerNode => IsActionNode && Owner == PlayerType.
        Attacker;
    public bool IsDefenderNode => IsActionNode && Owner == PlayerType.
        Defender;

    public override bool Equals(object obj)
    {
        return obj is Node other && Id == other.Id;
    }

    public override int GetHashCode()
    {
        return Id.GetHashCode();
    }
}
```

Listing 8.1: Node class in C#

```
smg

// Define players
player attacker
    Attacker, [attack_cameran9], [attack_eavesdropn5], [attack_forcen4], [
        attack_guessPWDn13], [attack_guessUsernamen18], [
        attack_logInExecuteTransfern19], [attack_Phishingn16], [
        attack_stealCardn2], [attack_stealPhonen23], [
        attack_withdrawCashn10], [attack_end_turn]
endplayer

player defender
```

```
9       Defender, [defense_coverKeypadn7], [defense_smsAuthn21], [
            defense_strongPWDn15], [defense_end_turn]
10  endplayer
11
12  player gameManager
13      GameManager, [check_gates]
14  endplayer
15
16
17  //Global variables
18  global turn : [0..3] init 0;
19  global top_event : bool init false;
20  //Attacker variables
21  global stealCard_n2: [0..1] init 0;
22  global force_n4: [0..1] init 0;
23  global eavesdrop_n5: [0..1] init 0;
24  global camera_n9: [0..1] init 0;
25  global withdrawCash_n10: [0..1] init 0;
26  global guessPWD_n13: [0..1] init 0;
27  global Phishing_n16: [0..1] init 0;
28  global guessUsername_n18: [0..1] init 0;
29  global logInExecuteTransfer_n19: [0..1] init 0;
30  global stealPhone_n23: [0..1] init 0;
31  //Defender variables
32  global coverKeypad_n7: [0..1] init 0;
33  global strongPWD_n15: [0..1] init 0;
34  global smsAuth_n21: [0..1] init 0;
35
36  global attacker_done : bool init false;
37  global defender_done : bool init false;
38
39
40
41  const int INIT_ATTACKER_BUDGET;
42  global attacker_budget: [0..645] init INIT_ATTACKER_BUDGET;
43  const int INIT_DEFENDER_BUDGET;
44  global defender_budget: [0..60] init INIT_DEFENDER_BUDGET;
45
46
47  formula n9_effective = (camera_n9=1);
48  formula cm_coverKeypad_n8_triggered = (coverKeypad_n7=1 & !n9_effective);
        //<----cm_coverKeypad
49  formula n8_effective = (cm_coverKeypad_n8_triggered);
50  formula cm_eavesdrop_n6_triggered = (eavesdrop_n5=1 & !n8_effective); //
        <----cm_eavesdrop
51  formula learnPIN_n3_triggered = (force_n4=1 | cm_eavesdrop_n6_triggered);
        //<----learnPIN
52  formula ATM_n1_triggered = (stealCard_n2=1 & learnPIN_n3_triggered &
        withdrawCash_n10=1); //<----ATM
53  formula n15_effective = (strongPWD_n15=1);
54  formula cm_guessPWD_n14_triggered = (guessPWD_n13=1 & !n15_effective); //
        <----cm_guessPWD
55  formula getPassword_n12_triggered = (cm_guessPWD_n14_triggered |
        Phishing_n16=1); //<----getPassword
56  formula getUserName_n17_triggered = (Phishing_n16=1 | guessUsername_n18=1);
         //<----getUserName
57  formula n23_effective = (stealPhone_n23=1);
58  formula cm_smsAuth_n22_triggered = (smsAuth_n21=1 & !n23_effective); //
        <----cm_smsAuth
59  formula n22_effective = (cm_smsAuth_n22_triggered);
60  formula cm_logInExecuteTransfer_n20_triggered = (logInExecuteTransfer_n19=1
```

```
      & !n22_effective); //<----cm_logInExecuteTransfer
61 formula viaOnlineBanking_n11_triggered = (getPassword_n12_triggered &
       getUserName_n17_triggered & cm_logInExecuteTransfer_n20_triggered); //
       <----viaOnlineBanking
62 formula stealFA_n0_triggered = (ATM_n1_triggered |
       viaOnlineBanking_n11_triggered); //<----stealFA
63
64 module Attacker
65     [attack_stealCardn2] turn=1 & stealCard_n2=0 & attacker_budget>=60 -> (
           stealCard_n2'=1) & (attacker_budget'=attacker_budget-60);
66     [attack_forcen4] turn=1 & force_n4=0 & attacker_budget>=100 -> (
           force_n4'=1) & (attacker_budget'=attacker_budget-100);
67     [attack_eavesdropn5] turn=1 & eavesdrop_n5=0 & attacker_budget>=20 -> (
           eavesdrop_n5'=1) & (attacker_budget'=attacker_budget-20);
68     [attack_cameran9] turn=1 & camera_n9=0 & attacker_budget>=75 -> (
           camera_n9'=1) & (attacker_budget'=attacker_budget-75);
69     [attack_withdrawCashn10] turn=1 & withdrawCash_n10=0 & attacker_budget
           >=10 -> (withdrawCash_n10'=1) & (attacker_budget'=attacker_budget
           -10);
70     [attack_guessPWDn13] turn=1 & guessPWD_n13=0 & attacker_budget>=120 ->
           (guessPWD_n13'=1) & (attacker_budget'=attacker_budget-120);
71     [attack_Phishingn16] turn=1 & Phishing_n16=0 & attacker_budget>=70 -> (
           Phishing_n16'=1) & (attacker_budget'=attacker_budget-70);
72     [attack_guessUsernamen18] turn=1 & guessUsername_n18=0 &
           attacker_budget>=120 -> (guessUsername_n18'=1) & (attacker_budget'=
           attacker_budget-120);
73     [attack_logInExecuteTransfern19] turn=1 & logInExecuteTransfer_n19=0 &
           attacker_budget>=10 -> (logInExecuteTransfer_n19'=1) & (
           attacker_budget'=attacker_budget-10);
74     [attack_stealPhonen23] turn=1 & stealPhone_n23=0 & attacker_budget>=60
           -> (stealPhone_n23'=1) & (attacker_budget'=attacker_budget-60);
75     [attack_end_turn] turn=1 -> (turn'=2) & (attacker_done'=true);
76 endmodule
77 module Defender
78     [defense_coverKeypadn7] turn=0 & coverKeypad_n7=0 & defender_budget>=30
            -> (coverKeypad_n7'=1) & (defender_budget'=defender_budget-30);
79     [defense_strongPWDn15] turn=0 & strongPWD_n15=0 & defender_budget>=10
           -> (strongPWD_n15'=1) & (defender_budget'=defender_budget-10);
80     [defense_smsAuthn21] turn=0 & smsAuth_n21=0 & defender_budget>=20 -> (
           smsAuth_n21'=1) & (defender_budget'=defender_budget-20);
81     [defense_end_turn] turn=0 -> (turn'=1) & (defender_done'=true);
82 endmodule
83 module GameManager
84     [check_gates] turn=2 -> (top_event'=stealFA_n0_triggered) & (turn'=3);
85 endmodule
86
87
88 rewards "attacker_cost"
89     [attack_stealCardn2] true : 60;
90     [attack_forcen4] true : 100;
91     [attack_eavesdropn5] true : 20;
92     [attack_cameran9] true : 75;
93     [attack_withdrawCashn10] true : 10;
94     [attack_guessPWDn13] true : 120;
95     [attack_Phishingn16] true : 70;
96     [attack_guessUsernamen18] true : 120;
97     [attack_logInExecuteTransfern19] true : 10;
98     [attack_stealPhonen23] true : 60;
99     [attack_end_turn] true : 0;
100 endrewards
101 rewards "defender_cost"
```

```
102     [defense_coverKeypadn7] true : 30;
103     [defense_strongPWDn15] true : 10;
104     [defense_smsAuthn21] true : 20;
105     [defense_end_turn] true : 0;
106 endrewards
107
108
109 label "top_event_reached" = top_event;
110 label "system_secure" = !top_event;
111 label "end_game" = (turn = 3);
```

Listing 8.2: PRISM code for ADT on Fig 6.1

Bibliography

[1] Aliyu Tanko Ali and Damas P Gruska. Attack trees with time constraints. In *CS&P*, pages 93–105, 2021.

[2] John Andrews and Silvia Tolo. Dynamic and dependent tree theory (d2t2): A framework for the analysis of fault trees with dependent basic events. *Reliability Engineering & System Safety*, 230:108959, 2023.

[3] Zaruhi Aslanyan, Flemming Nielson, and David Parker. Quantitative verification and synthesis of attack-defence scenarios. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 105–119. IEEE, 2016.

[4] Alessandra Bagnato, Barbara Kordy, Per Håkon Meland, and Patrick Schweitzer. Attribute decoration of attack–defense trees. *International Journal of Secure Software Engineering (IJSSE)*, 3(2):1–35, 2012.

[5] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM (JACM)*, 44(2):201–236, 1997.

[6] Andrea Bobbio, Lavinia Egidi, and Roberta Terruggia. A methodology for qualitative/quantitative analysis of weighted attack trees. *IFAC Proceedings Volumes*, 46(22):133–138, 2013.

[7] Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991.

[8] Carlos E Budde and Mariëlle Stoelinga. Efficient algorithms for quantitative attack tree analysis. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE, 2021.

[9] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, pages 359–364. Springer, 2002.

[10] Danut-Valentin Copae. Attack-defense trees with offensive and defensive attributes. Master's thesis, University of Twente, 2024.

[11] Danut-Valentin Copae, Reza Soltani, and Milan Lopuhaä-Zwakenberg. Attack-defense trees with offensive and defensive attributes (with appendix). *arXiv preprint arXiv:2504.12748*, 2025.

[12] Kalyanmoy Deb and Jayavelmurugan Sundar. Reference point based multi-objective optimization using evolutionary algorithms. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 635–642, 2006.

[13] Florian Dorfhuber, Julia Eisentraut, Katharina Klioba, and Jan Křetínský. Quadtool: Attack-defense-tree synthesis, analysis and bridge to verification. In *International Conference on Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems*, pages 52–71. Springer, 2024.

[14] Yassmeen Elderhalli, Osman Hasan, Waqar Ahmad, and Sofiène Tahar. Formal dynamic fault trees analysis using an integration of theorem proving and model checking. In *NASA Formal Methods Symposium*, pages 139–156. Springer, 2018.

[15] Barbara Fila and Wojciech Wideł. Efficient attack-defense tree analysis using pareto attribute domains. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 200–20015. IEEE, 2019.

[16] Marnie Grenat. Uppaal: An advanced tool for modeling and verification of real-time systems. https://dev.to/marniegrenat/uppaal-an-advanced-tool-for-modeling-and-verification-of-real-time-systems-a2j, 2023. Accessed: 2024-02-06.

[17] Sigrid Gürgens, Peter Ochsenschläger, and Carsten Rudolph. Abstractions preserving parameter confidentiality. In *Computer Security–ESORICS 2005: 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005. Proceedings 10*, pages 418–437. Springer, 2005.

[18] Till Hänisch and Christoph Karg. Estimating the success of it security measures in industry 4.0 environments using monte carlo simulation on attack defense trees. *Athens Journal of Technology & Engineering*, 6(4):211–222, 2019.

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

[20] Fatemeh Kazemeyni, Einar Broch Johnsen, Olaf Owe, and Ilangko Balasingham. Mule-based wireless sensor networks: probabilistic modeling and quantitative analysis. In *International Conference on Integrated Formal Methods*, pages 143–157. Springer, 2012.

[21] Arsalaan Khan. Optimizing security strategies: A game theoretic approach to attack defense trees. B.S. thesis, University of Twente, 2024.

[22] Barbara Kordy, Piotr Kordy, Sjouke Mauw, and Patrick Schweitzer. Adtool: security analysis with attack–defense trees. In *International conference on quantitative evaluation of systems*, pages 173–176. Springer, 2013.

[23] Barbara Kordy, Sjouke Mauw, Matthijs Melissen, and Patrick Schweitzer. Attack–defense trees and two-player binary zero-sum extensive form games are equivalent. In *Decision and Game Theory for Security: First International Conference, GameSec 2010, Berlin, Germany, November 22-23, 2010. Proceedings 1*, pages 245–256. Springer, 2010.

[24] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Foundations of attack–defense trees. In *Formal Aspects of Security and Trust: 7th International Workshop, FAST 2010, Pisa, Italy, September 16-17, 2010. Revised Selected Papers 7*, pages 80–95. Springer, 2011.

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

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

[27] Barbara Kordy and Wojciech Wideł. On quantitative analysis of attack–defense trees with repeated labels. In *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 7*, pages 325–346. Springer, 2018.

[28] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[29] Milan Lopuhaä-Zwakenberg, Carlos E Budde, and Mariëlle Stoelinga. Efficient and generic algorithms for quantitative attack tree analysis. *IEEE Transactions on Dependable and Secure Computing*, 20(5):4169–4187, 2022.

[30] Milan Lopuhaä-Zwakenberg and Mariëlle Stoelinga. Cost-damage analysis of attack trees. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 545–558. IEEE, 2023.

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

[32] Stefano M Nicoletti, Milan Lopuhaä-Zwakenberg, E Moritz Hahn, and Mariëlle Stoelinga. Querying fault and attack trees: Property specification on a water network. In *2024 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–6. IEEE, 2024.

[33] Stefano M Nicoletti, Milan Lopuhaä-Zwakenberg, Ernst Moritz Hahn, and Mariëlle Stoelinga. : A logic for quantitative security properties on attack trees. In *International Conference on Software Engineering and Formal Methods*, pages 205–225. Springer, 2023.

[34] Ali A Noroozi, Khayyam Salehi, Jaber Karimpour, and Ayaz Isazadeh. Secure information flow analysis using the prism model checker. In *Information Systems Security: 15th International Conference, ICISS 2019, Hyderabad, India, December 16–20, 2019, Proceedings 15*, pages 154–172. Springer, 2019.

[35] PRISM Model Checker. Prism - probabilistic model checking. https://www.prismmodelchecker.org/, 2024. Accessed: 2024-02-06.

[36] Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[37] Bruce Schneier. *Schneier on security.* John Wiley & Sons, 2009.

[38] NH Wolters. Analysis of attack trees with timed automata (transforming formalisms through metamodeling). Master's thesis, University of Twente, 2016.