Computational Aspects of Attack–Defense Trees

Barbara Kordy*, Marc Pouly, and Patrick Schweitzer**

CSC and SnT, University of Luxembourg,
6, rue Coudenhove–Kalergi, L–1359 Luxembourg
{barbara.kordy,marc.pouly,patrick.schweitzer}@uni.lu

Abstract. Attack–defense trees extend attack trees with defense nodes. This richer formalism allows for a more precise modeling of a system's vulnerabilities, by representing interactions between possible attacks and corresponding defensive measures. In this paper we compare the computational complexity of both formalisms. We identify semantics for which extending attack trees with defense nodes does not increase the computational complexity. This implies that, for these semantics, every query that can be solved efficiently on attack trees can also be solved efficiently on attack–defense trees. Furthermore, every algorithm for attack trees can directly be used to process attack–defense trees.

1 Introduction

Systems become more and more complex as technology is advancing faster and faster. This technological development goes along with more sophisticated attacks on systems. In 1999, Schneier [1] suggested attack trees as a visual method to evaluate the security of complex systems. An attack tree is an AND-OR structure detailing an attack scenario. Schneier advocated attack trees, but he was not the first to suggest such an approach. Weiss [2] and Amoroso [3] were two pioneers in the usage of trees in security analysis. But even as early as the 1960s, tree-like structures were used in risk analysis, see Vesely et al. [4]. In 2005, Mauw and Oostdijk [5] augmented attack trees with semantics, providing a solid, formal and methodological framework for security assessment. Since then, the attack tree methodology has been taken up by numerous researchers, see [6–11].

Attack trees are widely used to evaluate vulnerabilities of systems. However, there are several important aspects of security that they cannot model. Besides the fact that the attack tree formalism only considers an attacker's point of view, it can neither capture the interaction between an attacker and a defender, nor is it well-suited to depict the evolution of attacks and subsequent defenses.

To overcome these limitations, Kordy et al. recently extended the attack tree formalism with defensive measures, by introducing attack–defense trees, see [12]. A main difference between attack trees and attack–defense trees is that the latter allow for a more precise analysis of scenarios by repeatedly changing between

* Supported by grant No. C08/IS/26 from the National Research Fund, Luxembourg.
** Supported by grant No. PHD-09-167 from the National Research Fund, Luxembourg.

P. Bouvry et al. (Eds.): SIIS 2011, LNCS 7053, pp. 103–116, 2012.

an attacker's and a defender's perspective. Thus, the new formalism enlarges the modeling capabilities of attack trees. Moreover, attack–defense trees can be interpreted with several semantics, which allows us to deal with different facets of security. In particular, the choice of an appropriate semantics becomes essential when performing a quantitative analysis of an attack–defense scenario.

An especially important semantics for attack–defense trees is the propositional semantics. It is well-suited to answer feasibility questions, such as whether a system is vulnerable to an attack, whether special equipment is needed to perform an attack, or how many different ways of attacking exist. The propositional semantics has been studied in [13], where it was shown that satisfiability of an attack–defense tree is equivalent to the existence of a winning strategy in a two-player binary zero-sum game.

Our contribution. The main goal of [12] was to enrich the well-established attack tree model, but this work did not consider computational aspects. The following questions therefore remained unanswered:

- How hard is query evaluation on attack–defense trees with respect to query evaluation on attack trees?
- Are there queries that can efficiently be solved on attack trees but not on attack–defense trees?
- Are new algorithms needed to efficiently process attack–defense trees?

In the current paper, we address these problems for a large class of semantics for attack–defense trees. We prove that, when the propositional semantics is used, attack trees and attack–defense trees both represent monotone Boolean functions. Moreover, we show that the same holds if the semantics is induced by arbitrary De Morgan lattices. This lets us conclude that, for every semantics induced by a De Morgan lattice, enriching attack trees with defense nodes has not increased the computational complexity of the model. In particular, we argue that algorithms for attack trees can also be used to process attack–defense trees. Query evaluation on attack–defense trees is thus not harder than the corresponding query evaluation on attack trees. Hence, every query that can efficiently be solved on attack trees, can also efficiently be solved on attack–defense trees.

Structure. In Section 2, we recall basic definitions and introduce necessary notation. Section 3 proves our results for the propositional semantics. In Section 4, we show how to generalize them to semantics induced by De Morgan lattices. We discuss practical consequences and applications of our theoretical results in Section 5. Finally, Section 6 lists possible directions for future research.

2 Preliminaries

We start by recalling necessary facts about attack–defense trees, attack–defense terms and Boolean functions.

2.1 Attack–Defense Trees

Attack trees [1, 5] are a well-known methodology for assessing the security of complex systems. An attack tree is a rooted tree representing an attack scenario.

The root of an attack tree depicts the main goal of an attacker, and the other nodes constitute refinements of this goal into sub-goals. Two kinds of refinements are possible: conjunctive and disjunctive. A conjunctively refined (sub-)goal is satisfied if all its children are fulfilled, and a disjunctively refined (sub-)goal is satisfied when at least one of its children is fulfilled. The leaves of an attack tree represent basic actions which are used to build complex attacks.

Attack–defense trees [12] are attack trees extended with defense nodes. They represent attack–defense scenarios involving actions of an attacker trying to compromise a system and counteractions of a defender trying to protect the system. Consequently, an attack–defense tree can be seen as a game between two players: an attacker and a defender. Each node of an attack–defense tree depicts a (sub-)goal of one of the players, and the root node represents the main goal of an attacker or of a defender, depending on the modeler's perspective. Therefore, instead of talking about attacker and defender, we rather refer to them as *proponent* and *opponent*. By proponent we mean the player related to the root node, and by opponent we mean the other player. As in the case of attack trees, every node of an attack–defense tree can be refined in a conjunctive or a disjunctive way. The refinement is modeled using child nodes of the same type (proponent or opponent) as the type of the parent node. In addition, each node of an attack–defense tree may have one child of the opposite type. Such a child then represents a countermeasure that can be applied to counter or mitigate the (sub-)goal represented by its parent. Finally, every node without any child of the same type represents a basic action. Contrary to attack trees, such a node does not have to be a leaf, because it can still have a child of the opposite type.

2.2 Attack–Defense Terms

Attack–defense trees can formally be represented using so-called attack–defense terms. We briefly recall the construction of these terms and refer to [12], for a more detailed description.

Let $\mathcal{S} = \{\mathrm{p}, \mathrm{o}\}$ be a set of types representing a proponent and an opponent. Given a player $s \in \mathcal{S}$, we write $\overline{s}$ to denote the opposite player. By $\mathbb{B}$ we denote a set of constants called *basic actions*. The set of basic actions $\mathbb{B}$ is partitioned into the set of basic actions of the proponent's type, denoted by $\mathbb{B}^{\mathrm{p}}$, and the set of basic actions of the opponent's type, denoted by $\mathbb{B}^{\mathrm{o}}$. We use the typed operators $\vee^{\mathrm{p}}, \wedge^{\mathrm{p}}$ to model disjunctive and conjunctive refinements for the proponent and the corresponding operators $\vee^{\mathrm{o}}, \wedge^{\mathrm{o}}$ for the opponent. Moreover, to connect actions of one player with counteractions of the other player, we use the counter operators c^{p} and c^{o}.

Definition 1. *Attack–defense terms (ADTerms) are typed ground terms recursively constructed from* $\mathbb{B}$ *using the typed operators* $\vee^s, \wedge^s, \mathrm{c}^s$*, for* $s \in \mathcal{S}$*. The set of all ADTerms is denoted by* $\mathbb{T}$*.*

Given a player $s \in \mathcal{S}$, we say that an ADTerm t is *of type* s, if its head symbol is $\vee^s, \wedge^s, \mathrm{c}^s$, or if t is a constant from $\mathbb{B}^s$. The typed operators $\vee^s$ and $\wedge^s$ are unranked, i.e., they take an arbitrary number of terms of type s as arguments

and return a term of type s. The counter operator c^s is binary. It takes a term of type s as the first argument and a term of type $\overline{s}$ as the second argument, and returns a term of type s. By $\mathbb{T}^{\mathrm{p}}$ we denote the set of ADTerms of the proponent's type and by $\mathbb{T}^{\mathrm{o}}$ the set of ADTerms of the opponent's type. The ADTerms of the proponent's type constitute formal representations of attack–defense trees. Finally, the elements of $\mathbb{T}^{\mathrm{p}}$ which are built by using the operators $\vee^{\mathrm{p}}$ and $\wedge^{\mathrm{p}}$ only represent attack trees and are called *ATerms*.

Example 1. Let $a, b, d \in \mathbb{B}^{\mathrm{p}}$ be basic actions of the proponent's type and let $e, g \in \mathbb{B}^{\mathrm{o}}$ be basic actions of the opponent's type. The ADTerm

$$t = \wedge^{\mathrm{p}}(a, \mathrm{c}^{\mathrm{p}}(\vee^{\mathrm{p}}(b, d), \vee^{\mathrm{o}}(e, g)))$$

is of the proponent's type. It expresses a scenario in which, in order to achieve his goal, the proponent has to execute the action a and one of the actions b or d. At the same time, the opponent has the possibility to counter the proponent's actions b and d by executing at least one of the actions e or g. The opponent's ability of countering is indicated by the operator c^{p}, which takes the term $\vee^{\mathrm{p}}(b, d)$ of the proponent's type as the first argument and the term $\vee^{\mathrm{o}}(e, g)$ of the opponent's type as the second argument.

Different attack–defense trees (and therefore different ADTerms) may represent the same attack–defense scenario. Hence, we consider ADTerms modulo an equivalence relation.

Definition 2. *A semantics for ADTerms is an equivalence relation on* $\mathbb{T}$ *which preserves types.*

Every semantics partitions the set $\mathbb{T}$ into equivalence classes, and ADTerms belonging to the same equivalence class represent the same scenario.

Several distinct semantics for ADTerms have been introduced in [12]. One of them is the propositional semantics discussed in the following section.

2.3 Propositional Semantics for ADTerms

The idea behind the propositional semantics for ADTerms is to first associate a propositional formula with every ADTerm and then deduce an equivalence relation on $\mathbb{T}$ from the canonical equivalence relation of propositional logic.

In this paper, r denotes a countable set of propositional variables. First, with every basic action $b \in \mathbb{B}$, we associate a propositional variable $X_b \in r$. We assume that for $b, b' \in \mathbb{B}$, with $b \neq b'$, we have $X_b \neq X_{b'}$. In particular, since the sets of basic actions of the proponent's and of the opponent's type are disjoint, we have

$$\{X_b \mid b \in \mathbb{B}^{\mathrm{p}}\} \cap \{X_b \mid b \in \mathbb{B}^{\mathrm{o}}\} = \emptyset.$$

Second, a propositional formula $t_{\mathcal{P}}$ is associated with every ADTerm t, as follows

$$t_{\mathcal{P}} = \begin{cases} X_b, & \text{if } t = b \in \mathbb{B}, \\ t^1_{\mathcal{P}} \vee \cdots \vee t^k_{\mathcal{P}}, & \text{if } t = \vee^s_k(t^1, \ldots, t^k), \\ t^1_{\mathcal{P}} \wedge \cdots \wedge t^k_{\mathcal{P}}, & \text{if } t = \wedge^s_k(t^1, \ldots, t^k), \\ t^1_{\mathcal{P}} \wedge \neg t^2_{\mathcal{P}}, & \text{if } t = \mathrm{c}^s(t^1, t^2), \end{cases}$$

where $s \in \{\mathrm{p}, \mathrm{o}\}$ and $k \in \mathbb{N}$. A formula $t_{\mathcal{P}}$ is referred to as *propositional ADTerm*. In case of an ATerm, the corresponding formula is called a *propositional ATerm*.

By $\approx$ we denote the equivalence relation on propositional formulæ. Recall that two propositional formulæ ψ and ψ' are equivalent ($\psi \approx \psi'$) if and only if, for every assignment ν of Boolean values to variables in r, we have $\nu(\psi) = \nu(\psi')$.

Definition 3. *The propositional semantics for ADTerms is the equivalence relation $\equiv_{\mathcal{P}}$ on $\mathbb{T}$ defined, for all $t, t' \in \mathbb{T}$, by $t \equiv_{\mathcal{P}} t'$ if and only if $t_{\mathcal{P}} \approx t'_{\mathcal{P}}$.*

2.4 Boolean Functions

The set of propositional ADTerms can be seen as a representation language for Boolean functions. The analysis of this language, performed in Section 3, allows us to compare the computational complexity of ADTerms and ATerms. The current section gathers necessary definitions and facts about Boolean functions.

A *configuration* with finite domain $d \subseteq r$ is a function $\mathbf{x}\colon d \to \{0,1\}$ that associates a value $\mathbf{x}(X) \in \{0,1\}$ with every variable $X \in d$. Thus, a configuration $\mathbf{x} \in \{0,1\}^d$ represents an assignment of Boolean values to the variables in d.

Definition 4. *A Boolean function f with domain d is a function $f\colon \{0,1\}^d \to \{0,1\}$ that assigns a value $f(\mathbf{x}) \in \{0,1\}$ to each configuration $\mathbf{x} \in \{0,1\}^d$.*

Given a configuration $\mathbf{x}$ with domain $d \subseteq r$, we denote by $\mathbf{x}^{\downarrow u}$ the projection of $\mathbf{x}$ to a subset $u \subseteq d$. This notation allows us to introduce the following definition.

Definition 5. *Let f and g be two Boolean functions with domains d and u, respectively. The conjunction ($f \wedge g$) and the disjunction ($f \vee g$) of f and g are Boolean functions with domain $d \cup u$, defined for every $\mathbf{x} \in \{0,1\}^{d \cup u}$ by*

$$f \wedge g(\mathbf{x}) = \min\{f(\mathbf{x}^{\downarrow d}), g(\mathbf{x}^{\downarrow u})\}, \qquad f \vee g(\mathbf{x}) = \max\{f(\mathbf{x}^{\downarrow d}), g(\mathbf{x}^{\downarrow u})\}.$$

The negation of f, denoted by $\neg f$, is a Boolean function with domain d, defined for every $\mathbf{x} \in \{0,1\}^d$ by $(\neg f)(\mathbf{x}) = 1 - f(\mathbf{x})$.

Let $u \subseteq r$ be a finite set of propositional variables. By $e_u\colon \{0,1\}^u \to \{0,1\}$ we denote the Boolean unit function, i.e., $e_u(\mathbf{x}) = 1$, for every $\mathbf{x} \in \{0,1\}^u$. Given a Boolean function f with domain d, we denote by $f^{\uparrow d \cup u}$ the *vacuous extension of f to $d \cup u$*, defined as $f^{\uparrow d \cup u} = f \wedge e_{u \setminus d}$.

Definition 6. *Two Boolean functions f and g, with respective domains d and u, are said to be equivalent (denoted by $f \equiv g$) if and only if, $\forall \mathbf{x} \in \{0,1\}^{d \cup u}$, we have $f^{\uparrow d \cup u}(\mathbf{x}) = g^{\uparrow d \cup u}(\mathbf{x})$.*

As shown in [14], $f \equiv g$ if and only if $\forall \mathbf{x} \in \{0,1\}^r$, $f^{\uparrow r}(\mathbf{x}) = g^{\uparrow r}(\mathbf{x})$. The advantage of an equivalence relation using finite sets of variables, as in Definition 6, is that the construction described in this paper is practical and implementable.

Remark 1. Since equivalent propositional formulæ represent equivalent Boolean functions, two ADTerms t and t' are equivalent under the propositional semantics, if they represent equivalent Boolean functions.

Of particular importance for our studies are positive, negative and monotone Boolean functions.

Definition 7. *Let f be a Boolean function with domain $d \subseteq r$, and let $X \in d$ be a propositional variable.*

- *f is positive in X if $f(\mathbf{x}, 0) \leq f(\mathbf{x}, 1)$, for all $\mathbf{x} \in \{0,1\}^{d \setminus \{X\}}$,*
- *f is negative in X if $f(\mathbf{x}, 0) \geq f(\mathbf{x}, 1)$, for all $\mathbf{x} \in \{0,1\}^{d \setminus \{X\}}$,*
- *f is monotone in $X \in d$ if it is either positive or negative in X.*

Note that if $X \in r$ does not occur in the domain of a Boolean function f, then f is insensitive to the values assigned to X. In this case, we may say that f is positive, negative and monotone in X.

Definition 8. *A Boolean function f is positive (resp. negative, monotone) if it is positive (resp. negative, monotone) in every variable $X \in r$.*

The following lemma shows that the classes of positive, as well as negative Boolean functions are closed under conjunction and disjunction.

Lemma 1. *Let f and g be Boolean functions,*

- *if f and g are positive in X, then $f \wedge g$ and $f \vee g$ are positive in X,*
- *if f and g are negative in X, then $f \wedge g$ and $f \vee g$ are negative in X.*

Proof. Both statements follow directly from the monotonicity of minimization and maximization. □

Note however, that the results from Lemma 1 do generally not hold for monotone Boolean functions.

Example 2. The Boolean function $f(X, Y) = X \wedge \neg Y$ is positive in X and negative in Y. Thus, f is monotone. For similar reasons, the Boolean function $g(X, Y) = Y \wedge \neg X$ is monotone. However, it can easily be checked that the function $f \vee g$ is not monotone.

Next we show that monotone Boolean functions are closed under negation.

Lemma 2. *Let f be a Boolean function, and let $X \in r$ be a variable. If f is positive (resp. negative) in X, then $\neg f$ is negative (resp. positive) in X.*

Proof. Let us assume that f is positive in X, and let the domain of f be denoted by d. If $X \notin d$, then, by convention, f is positive in X and $\neg f$ is negative in X. If $X \in d$, then from the positivity of f in X, we have $f(\mathbf{x}^{\downarrow d \setminus \{X\}}, 0) \leq f(\mathbf{x}^{\downarrow d \setminus \{X\}}, 1)$, for all $\mathbf{x} \in \{0,1\}^d$. Therefore,

$$(\neg f)(\mathbf{x}^{\downarrow d \setminus \{X\}}, 0) = 1 - f(\mathbf{x}^{\downarrow d \setminus \{X\}}, 0) \geq 1 - f(\mathbf{x}^{\downarrow d \setminus \{X\}}, 1) = (\neg f)(\mathbf{x}^{\downarrow d \setminus \{X\}}, 1).$$

This shows that $\neg f$ is negative in X. The proof for the other case is similar. □

Note that Lemma 2 holds because negation $\neg$ reverses the order, i.e., for $a, b \in \{0, 1\}$, we have $a \leq b$ if and only if $\neg a \geq \neg b$. This is crucial in Section 4, where we generalize this result from the propositional algebra to De Morgan lattices.

From Lemmas 1 and 2, we deduce the following result.

Corollary 1. *If f and g are two Boolean functions, such that f is positive (resp. negative) in a variable X and g is negative (resp. positive) in X, then the Boolean function $f \wedge \neg g$ is positive (resp. negative) in X.*

3 Transformation of ADTerms to ATerms

The objective of this section is to compare the computational complexity of the propositional ADTerms language with the computational complexity of the propositional ATerms language. We achieve this by analyzing the classes of Boolean functions represented by both languages.

3.1 Expressiveness of Propositional ADTerms

We start by analyzing the language of propositional ATerms. ATerms constitute formal representations of attack trees, which are AND-OR trees containing only proponent's nodes. Therefore, every propositional ATerm is a formula generated by the following grammar $\mathcal{AT}$

$$P: \quad X^{\mathrm{p}} \quad | \quad P \vee P \quad | \quad P \wedge P, \tag{$\mathcal{AT}$}$$

where $X^{\mathrm{p}} \in \{X_b \mid b \in \mathbb{B}^{\mathrm{p}}\}$. Theorem 1 characterizes propositional ATerms using Boolean functions.

Theorem 1. *Boolean functions represented by propositional ATerms are positive.*

Proof. Consider the grammar $\mathcal{AT}$. The Boolean function represented by X^{p} is positive. The positivity of the Boolean functions represented by $P \vee P$ and $P \wedge P$ is a direct consequence of Lemma 1. □

In order to characterize the language of propositional ADTerms, we extend the grammar $\mathcal{AT}$ to the grammar $\mathcal{ADT}$ generating all propositional ADTerms

$$\begin{array}{lllllllll} P: & X^{\mathrm{p}} & | & P \vee P & | & P \wedge P & | & P \wedge \neg N \\ N: & X^{\mathrm{o}} & | & N \vee N & | & N \wedge N & | & N \wedge \neg P, \end{array} \tag{$\mathcal{ADT}$}$$

where $X^{\mathrm{p}} \in \{X_b \mid b \in \mathbb{B}^{\mathrm{p}}\}$ and $X^{\mathrm{o}} \in \{X_b \mid b \in \mathbb{B}^{\mathrm{o}}\}$. The formulæ of the form P (resp. N) are propositional ADTerms for the terms of the proponent's (resp. opponent's) type. Theorem 2 characterizes propositional ADTerms using Boolean functions.

Theorem 2. *Boolean functions represented by propositional ADTerms are monotone.*

In order to prove Theorem 2, we use the following Lemma.

Lemma 3. *Consider the grammar $\mathcal{ADT}$. Every Boolean function represented by a formula of the form P (resp. N) is*

- *positive (resp. negative) in every variable X_b, for $b \in \mathbb{B}^{\mathrm{p}}$,*
- *negative (resp. positive) in every variable X_b, for $b \in \mathbb{B}^{\mathrm{o}}$.*

Proof. We provide a proof in the case of P. A proof for N is analogous. We reason by induction over the structure of P. If $P = X^{\mathrm{p}} \in \mathbb{B}^{\mathrm{p}}$, then the Boolean function represented by P is clearly positive in X^{p}. According to our convention, P is also positive in every $X_b \in \mathbb{B}^{\mathrm{p}} \setminus \{X^{\mathrm{p}}\}$ and negative in every $X_b \in \mathbb{B}^{\mathrm{o}}$.

Now, consider a formula P which is not a single variable, and assume that the lemma holds for all formulæ composing P. If P is of the form $P \vee P$ or $P \wedge P$, the result follows from Lemma 1 and the induction hypothesis. If P is of the form $P \wedge \neg N$, the result follows from Corollary 1 and the induction hypothesis. □

Since the sets $\mathbb{B}^{\mathrm{p}}$ and $\mathbb{B}^{\mathrm{o}}$ are disjoint, we can conclude that every formula generated by $\mathcal{ADT}$ represents a monotone Boolean function. This proves Theorem 2. Note that the assumption that $\mathbb{B}^{\mathrm{p}}$ and $\mathbb{B}^{\mathrm{o}}$ are disjoint is crucial. Without this assumption, Lemma 3 would not hold, see Example 2.

3.2 From Propositional ADTerms to Propositional ATerms

Since every ATerm is also an ADTerm, it is obvious that all Boolean functions represented by propositional ATerms can also be represented by propositional ADTerms. In this section, we show that the converse holds as well.

Theorem 3. *Let f be a Boolean function with domain d and let $X \in d$. We define a Boolean function g with the same domain, as follows*

$$g(\mathbf{x}, 0) = f(\mathbf{x}, 1) \quad \textit{and} \quad g(\mathbf{x}, 1) = f(\mathbf{x}, 0),$$

for all $\mathbf{x} \in \{0,1\}^{d \setminus \{X\}}$. The function g is positive (resp. negative) in X if and only if the function f is negative (resp. positive) in X.

Proof. If f is positive in $X \in d$, then, for all $\mathbf{x} \in \{0,1\}^{d \setminus \{X\}}$, we have $g(\mathbf{x}, 1) = f(\mathbf{x}, 0) \leq f(\mathbf{x}, 1) = g(\mathbf{x}, 0)$, i.e., g is negative in X. The other case is similar. □

Note that the functions f and g in Theorem 3 are not equivalent in the sense of Definition 6, but there is a one-to-one correspondence between their satisfying assignments, i.e., between the elements of the sets of $f^{-1}(\{1\})$ and $g^{-1}(\{1\})$.

It follows from Theorem 3 that every monotone Boolean function, which is not positive, can always be transformed to a positive form. Moreover, Lemma 3 guarantees that such a transformation is linear with respect to the size of the function's domain. Consequently, whenever we want to reason about a propositional ADTerm, we can analyze a positive Boolean function instead of a monotone one. Hence, the following result holds.

Corollary 2. *Propositional ADTerms represent positive Boolean functions.*

This proves that the language of propositional ADTerms and the language of propositional ATerms both represent positive Boolean functions. Practical consequences of this fact are discussed in Section 5.

4 Generalizations

An important feature of ADTerms is that they can be equipped with different semantics. This allows for the analysis of various security aspects. However, the previous sections focus on the propositional semantics, where basic actions are allowed to take propositional values, and ADTerms over the set of variables d represent Boolean functions of the form $\{0,1\}^d \to \{0,1\}$. Hence, the propositional semantics is a semantics induced by the Boolean algebra $\langle \{0,1\}, \wedge, \vee, \neg \rangle$. In this section we show that the transformation from ADTerms to ATerms, presented in Section 3 for the propositional semantics, applies to all semantics induced by more general algebraic structures $\langle A, +, \times, \neg \rangle$, such as De Morgan lattices.

Let $\langle A, +, \times, \neg \rangle$ be an algebraic structure defined over a non-empty set A with two binary operations, $+$ and $\times$, and a unary operation $\neg$. Since we still consider propositional variables, ADTerms and ATerms now represent functions of the form $\{0,1\}^d \to A$. From the isomorphism property of Boolean algebras, we directly obtain the following corollary.

Corollary 3. *For every semantics induced by a finite Boolean algebra $\langle A, +, \times, \neg \rangle$, ADTerms can efficiently be transformed to ATerms.*

In the rest of this section, we show that the transformation even works with less algebraic structure. To distinguish between positive, negative and monotone functions of the form $\{0,1\}^d \to A$, over the structure $\langle A, +, \times, \neg \rangle$, the set A must exhibit a partial order that behaves monotonically under the operations $+$ and $\times$. It is well-known that if $\langle A, +, \times \rangle$ is a lattice it can always be equipped with a canonical partial order, defined for all $a, b \in A$, by

$$a \preceq b \text{ if and only if } a + b = b. \qquad (\preceq)$$

This order is monotonic with respect to the operations $+$ and $\times$, see [15]. This allows us to generalize Lemma 1. We then extend $\langle A, +, \times \rangle$ with a negation operation which reverses the order $\preceq$.

Definition 9. *A tuple $\langle A, +, \times, \neg \rangle$ is called a De Morgan lattice if $\langle A, +, \times \rangle$ is a distributive lattice and, for all $a, b \in A$, we have*

$$\neg(a + b) = (\neg a) \times (\neg b), \qquad \neg(a \times b) = (\neg a) + (\neg b), \qquad \neg(\neg a) = a.$$

To validate Lemma 2, we show that in De Morgan lattices the order $\preceq$ is indeed reversed under negation.

Lemma 4. *In a De Morgan lattice we have $a \preceq b$ if and only if $\neg b \preceq \neg a$.*

Proof. Assume that $a \preceq b$, i.e., $a + b = b$. It follows from the definition of a De Morgan lattice that $\neg b = \neg(a + b) = (\neg a) \times (\neg b)$. Moreover, in every lattice we have $b = a \times b$ if and only if $a = a + b$, see [15]. Therefore, we conclude that $\neg b = (\neg a) \times (\neg b)$ if and only if $\neg a = (\neg a) + (\neg b)$. This proves that $a \preceq b$ implies $\neg b \preceq \neg a$. Conversely, assume $\neg b \preceq \neg a$. From the first part of this proof we know that $\neg b \preceq \neg a$ implies $\neg(\neg a) \preceq \neg(\neg b)$, and therefore we have $a \preceq b$. □

Observe that De Morgan lattices are more general than Boolean algebras, because the former do not have to satisfy the law of the excluded middle and the law of non-contradiction.

Example 3. The tuple $\langle [0,1], \max, \min, \neg \rangle$, where $\neg a = 1 - a$, for every $a \in [0,1]$, is a De Morgan lattice which is not a Boolean algebra.

Consider a De Morgan lattice $\langle A, +, \times, \neg \rangle$ and a finite set $d \subseteq r$ of variables. A *De Morgan valuation* with domain d is a function of the form $f\colon \{0,1\}^d \to A$. Note that De Morgan valuations satisfy the properties of valuation algebras, see [16]. Similar to Definition 6, equivalence of De Morgan valuations is again defined by point-wise equality of their vacuous extensions. Furthermore, we obtain positive, negative and monotone De Morgan valuations by modifying Definition 7: we replace Boolean functions by De Morgan valuations and the order $\leq$ by $\preceq$.

The above considerations guarantee that the statements of Corollary 1 and Theorem 3 still hold if Boolean functions are replaced by De Morgan valuations. This means that the transformation of ADTerms to ATerms, described in Section 3 for the propositional semantics, actually holds for a larger class of semantics that we define below.

Let $\langle A, +, \times, \neg \rangle$ be a De Morgan lattice. As in the case of the propositional semantics, we consider the set of propositional variables $\{X_b \mid b \in \mathbb{B}\}$. With every ADTerm t we associate a De Morgan valuation f_t, called a *De Morgan ADTerm*, as follows. If $t = b$ and b is a basic action, then f_t is a function of the form $f_b\colon \{0,1\}^{\{X_b\}} \to A$. With the help of f_b, we express how the value assigned to a basic action b changes depending on whether the basic action b is satisfied ($X_b = 1$) or not ($X_b = 0$). De Morgan ADTerms associated with composed ADTerms are then defined recursively, as follows. For $s \in \{\mathrm{p}, \mathrm{o}\}$, $k \in \mathbb{N}$, we set [1]

$$f_{\vee^s(t_1,\ldots,t_k)} = \sum_{i=1}^{k} f_{t_i}, \qquad f_{\wedge^s(t_1,\ldots,t_k)} = \prod_{i=1}^{k} f_{t_i}, \qquad f_{\mathrm{c}^s(t_1,t_2)} = f_{t_1} \times \neg f_{t_2}.$$

Definition 10. *The semantics for ADTerms induced by the De Morgan lattice $\langle A, +, \times, \neg \rangle$ is the equivalence relation $\equiv_{\mathcal{DM}}$ on $\mathbb{T}$ defined, for all $t, t' \in \mathbb{T}$, by $t \equiv_{\mathcal{DM}} t'$ if and only if the corresponding De Morgan valuations f_t and $f_{t'}$ are equivalent.*

Before presenting the main result of this section, we briefly compare the usability of semantics induced by De Morgan lattices with the propositional semantics. Every Boolean algebra is a De Morgan lattice. Thus, the propositional semantics is the semantics induced by $\langle \{0,1\}, \vee, \wedge, \neg \rangle$, where a basic action b is interpreted as the Boolean function $f_b(X_b) = X_b$. Such a propositional interpretation implies that each action which is present is fully feasible, i.e., $f_b(X_b = 1) = 1$. This shows that the use of Boolean functions is not appropriate when one wants to describe the components of an attack–defense tree with more fine grained

[1] $\sum$ and $\prod$ stand for extensions of sum and product of two valuations to any finite number of valuations. They are correctly defined, due to associativity of $+$ and $\times$.

feasibility levels, such as, fully feasible (T), partially feasible (M) and infeasible (F), for instance. In particular, the propositional semantics cannot be applied to determine up to which level a proposed scenario is actually feasible. However, such a more detailed analysis can be performed using the semantics induced by the De Morgan lattice $\langle\{T, M, F\}, \max, \min, \neg\rangle$, where $F < M < T$ and $\neg F = T$, $\neg M = M$ and $\neg T = F$, as shown is Example 4.

Example 4. Consider the ADTerm $t = \mathrm{c}^{\mathrm{p}}(b, \wedge^{\mathrm{o}}(d, e))$ and the semantics induced by the De Morgan lattice $\langle\{T, M, F\}, \max, \min, \neg\rangle$. We assume that when the actions b, d and e are not present, they are infeasible

$$f_b(X_b = 0) = F, \qquad f_d(X_d = 0) = F, \qquad f_e(X_e = 0) = F.$$

Moreover, the presence of the actions b and e ensures their full feasibility, but the presence of the action d guarantees its partial feasibility, only

$$f_b(X_b = 1) = T, \qquad f_d(X_d = 1) = M, \qquad f_e(X_e = 1) = T.$$

Analyzing the De Morgan valuation associated with t, given by $f_t(X_b, X_d, X_e) = \min\{f_b(X_b), \neg(\min\{f_d(X_d), f_e(X_e)\})\}$, allows us to reason about feasibility of the scenario represented by t. We have

$$\begin{array}{llll} f_t(0,0,0) = F & f_t(0,1,0) = F & f_t(1,0,0) = T & f_t(1,1,0) = T \\ f_t(0,0,1) = F & f_t(0,1,1) = F & f_t(1,0,1) = T & f_t(1,1,1) = M. \end{array}$$

From $f_t^{-1}(\{M, T\})$, we deduce that the scenario is at least partially feasible for the proponent if the action b is present, independently of the actions d and e.

The considerations described in this section imply the following theorem.

Theorem 4. *De Morgan ADTerms are positive De Morgan valuations.*

This proves that the results obtained in Section 3 for the propositional semantics, generalize to every semantics induced by a De Morgan lattice.

5 Consequences

As discussed in [12], an obvious limitation of attack trees is that they cannot capture the interaction between attacks on a system and the defenses put in place to mitigate the attacks. To surpass this limitation, attack–defense trees have been developed. By allowing alternation between attack and defense nodes, attack–defense trees take the effects of existing defensive measures into account and allow us to consider the evolution of a system's security. In contrast to this, the results in Sections 3 and 4 compare attack and attack–defense trees on the computational level. We formally proved that, under a semantics induced by a De Morgan lattice, both models represent positive valuations and therefore exhibit the same computational complexity.

Result 1. *Attack–defense trees extend attack trees to a richer model without increasing the computational complexity, provided that their semantics is induced by a De Morgan lattice.*

Result 1 can be applied, for instance, to query evaluation on ADTerms. A *query* on ADTerms is a function $Q\colon \mathbb{T} \to \mathcal{A}$ which assigns to every ADTerm t an element $Q(t) \in \mathcal{A}$ called the *answer* for Q on t.

Example 5. Consider a function $Q_{\mathrm{SAT}}\colon \mathbb{T} \to \{\top, \bot\}$ which assigns $\top$ to an ADTerm t if the corresponding Boolean function f admits at least one satisfying assignment, i.e., if there exists a configuration $\mathbf{x}$, such that $f(\mathbf{x}) = 1$. Otherwise, $\bot$ is assigned to t. The function Q_{SAT} is an example of a query on ADTerms. It models satisfiability check, when the propositional semantics is used.

Result 2 characterizes how hard query evaluation is on ADTerms with respect to query valuation on ATerms.

Result 2. *When a semantics induced by a De Morgan lattice is used, the complexity of query evaluation on ADTerms is the same as the corresponding complexity for ATerms.*

It follows from Result 2 that, when a semantics induced by a De Morgan lattice is used, a query can efficiently be solved on ADTerms if and only if it can efficiently be solved on ATerms. As an example, when the propositional is used, satisfiability check on ADTerms can be performed in constant time, because all positive Boolean functions are satisfiable.

Theorem 3 and subsequent considerations in Section 4 show that, for a large class of semantics, we can effectively transform ADTerms to ATerms. Therefore, we obtain the following result.

Result 3. *When using a semantics induced by a De Morgan lattice, ADTerms can always be processed by algorithms developed for ATerms.*

Moreover, our constructive transformation guarantees that Result 3 holds for all existing and all future algorithms for ATerms.

Finally, knowing that not all Boolean functions are positive and taking into account Corollary 2, we deduce that there exist Boolean functions which cannot be represented by any propositional ADTerm.

Result 4. *The propositional language defined by propositional ADTerms is not complete.*

For instance, there is no ADTerm corresponding to the Boolean function representing the tautology. This example also shows that the set of propositional ADTerms represents a proper subset of the set of positive Boolean functions.

6 Open Problems

One of the particularities of positive Boolean functions is that they admit a unique modulo associativity and commutativity (AC), complete and irredundant

DNF representation, see [17]. This means that no other DNF representation of a positive Boolean function can be as short as its complete DNF, i.e., the disjunction of all its prime implicants. We can therefore conclude that ADTerms under the propositional semantics possess unique modulo AC normal forms. We believe that this provides a sufficient argument to find a finite axiomatization of the propositional semantics for ADTerms. This would be a first step towards an efficient implementation of an attack–defense tree tool that captures all semantics preserving transformations and is able to execute equivalence check between ADTerms. The importance of such transformations has been pointed out by Mauw and Oostdijk in [5].

Furthermore, we would like to know whether it is possible to apply our results to other classes of semantics. Of particular interests would be to replace propositional variables with multi-state variables, as in [18]. The advantage of multi-state variables is that they can express security levels more accurately than propositional variables. Using them we can, for instance, model nominal values or categories, such as *high*, *medium* and *low*.

We also plan to investigate the relation between modeling capabilities and computational complexity of attack–defense trees under the multiset semantics, introduced in [12]. The importance of this semantics arises from its compatibility with a multitude of attributes, such as the cost of the cheapest attack, the maximal damage caused by an attack or the probability of a successful attack.

Finally, we would like to take a look at other languages for representing Boolean functions. The topic has exhaustively been studied by Darwiche and Marquis. In [19], they established a taxonomy of complete propositional languages, based on the succinctness of the representation, the queries that can efficiently be answered on a given representation and the transformations that can be applied to a given language in polynomial time. Their paper considers several queries, including satisfiability and validity checks, model counting, model and counter-model enumeration. In the spirit of this taxonomy, we would like to answer runtime questions for the corresponding queries on our incomplete language of propositional ADTerms.

7 Conclusion

In this paper we perform an exhaustive analysis of a wide class of semantics for ADTerms. First, by employing known results from propositional logics, we prove that propositional ADTerms and propositional ATerms represent the same class of Boolean functions. We then show that this result can be generalized from Boolean functions to De Morgan valuations. We deduce that, for every semantics induced by a De Morgan lattice, the computational complexity of the attack tree model and the attack–defense tree model is the same. This proves that enriching the attack tree formalism with defense nodes was not done at the expense of computational complexity.

We also discuss several important consequences that can be derived for ADTerms interpreted using De Morgan valuations. In particular, we deduce that

the complexity of query evaluation on ADTerms is the same as the corresponding complexity on ATerms. Moreover, by showing that ADTerms can efficiently be transformed to ATerms, we conclude that algorithms for ATerms can be used to reason about ADTerms.

References

1. Schneier, B.: Attack Trees. Dr. Dobb's Journal of Software Tools 24(12), 21–29 (1999)
2. Weiss, J.D.: A system security engineering process. In: 14th Nat. Comp. Sec. Conf., pp. 572–581 (1991)
3. Amoroso, E.G.: Fundamentals of Computer Security Technology. Prentice-Hall, Inc., Upper Saddle River (1994)
4. Vesely, W.E., Goldberg, F.F., Roberts, N., Haasl, D.: Fault Tree Handbook. Technical Report NUREG-0492, U.S. Regulatory Commission (1981)
5. Mauw, S., Oostdijk, M.: Foundations of Attack Trees. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 186–198. Springer, Heidelberg (2006)
6. Cervesato, I., Meadows, C.: One Picture Is Worth a Dozen Connectives: A Fault-Tree Representation of NPATRL Security Requirements. IEEE TDSC 4, 216–227 (2007)
7. Edge, K.S., Dalton II, G.C., Raines, R.A., Mills, R.F.: Using Attack and Protection Trees to Analyze Threats and Defenses to Homeland Security. In: MILCOM, IEEE, pp. 1–7 (2006)
8. Morais, A.N.P., Martins, E., Cavalli, A.R., Jimenez, W.: Security Protocol Testing Using Attack Trees. In: CSE (2), pp. 690–697. IEEE Computer Society (2009)
9. Jürgenson, A., Willemson, J.: Serial Model for Attack Tree Computations. In: Lee, D., Hong, S. (eds.) ICISC 2009. LNCS, vol. 5984, pp. 118–128. Springer, Heidelberg (2010)
10. Bistarelli, S., Peretti, P., Trubitsyna, I.: Analyzing Security Scenarios Using Defence Trees and Answer Set Programming. ENTCS 197(2), 121–129 (2008)
11. Yager, R.R.: OWA trees and their role in security modeling using attack trees. Inf. Sci. 176(20), 2933–2959 (2006)
12. Kordy, B., Mauw, S., Radomirović, S., Schweitzer, P.: Foundations of Attack–Defense Trees. In: Degano, P., Etalle, S., Guttman, J. (eds.) FAST 2010. LNCS, vol. 6561, pp. 80–95. Springer, Heidelberg (2011)
13. Kordy, B., Mauw, S., Melissen, M., Schweitzer, P.: Attack–Defense Trees and Two-Player Binary Zero-Sum Extensive Form Games Are Equivalent. In: Alpcan, T., Buttyán, L., Baras, J.S. (eds.) GameSec 2010. LNCS, vol. 6442, pp. 245–256. Springer, Heidelberg (2010)
14. Kohlas, J.: Information Algebras: Generic Structures for Inference. Springer, Heidelberg (2003)
15. Davey, B., Priestley, H.: Introduction to Lattices and Order. Cambridge University Press (1990)
16. Pouly, M., Kohlas, J.: Generic Inference: A Unifying Theory for Automated Reasoning. John Wiley & Sons, Inc. (2011)
17. Crama, Y., Hammer, P.: Boolean Functions: Theory, Algorithms and Applications. Cambridge University Press (2011)
18. Wachter, M., Haenni, R.: Multi-state Directed Acyclic Graphs. In: Kobti, Z., Wu, D. (eds.) Canadian AI 2007. LNCS (LNAI), vol. 4509, pp. 464–475. Springer, Heidelberg (2007)
19. Darwiche, A., Marquis, P.: A Knowledge Compilation Map. J. Artif. Intell. Res. 17, 229–264 (2002)