Synthesising attack trees with optimal shape and labelling[a]

Olga Gadyatskaya[1][b], Sjouke Mauw[2][c], Rolando Trujillo-Rasua[3][d], Tim A. C. Willemse[4][e]
[1]*Leiden University, Leiden, The Netherlands*
[2]*DCS, University of Luxembourg, Esch-sur-Alzette, Luxembourg*
[3]*Universitat Rovira i Virgili, Tarragona, Spain*
[4]*Eindhoven University of Technology, Eindhoven, The Netherlands*
{*olga.gadyatskaya*}*@liacs.leidenuniv.nl, sjouke.mauw@uni.lu, rolando.trujillo@urv.cat, t.a.c.willemse@tue.nl*

Keywords: Attack trees, risk management, threat modelling, synthesis problem, algebraic factorisation

Abstract: This article addresses the problem of automatically generating attack trees that soundly and clearly describe the ways a system can be attacked. Soundness means that the attacks displayed by the attack tree are indeed attacks in the system; clarity means that the tree is efficient in communicating the attack scenario. To pursue clarity, we introduce an attack-tree synthesis algorithm that decorates trees with succinct and informative labels while minimizing their size. We achieve this by i) introducing a system model that allows to reason about attacks and goals in an efficient manner, and ii) by establishing a connection between the problem of factorising algebraic expressions and the problem of minimizing the tree size. To the best of our knowledge, we introduce the first attack-tree synthesis framework that optimises the labelling and shape of the generated trees, while guaranteeing their soundness with respect to a system specification.

1 Introduction

Attack trees, introduced by Schneier (Schneier, 1999), are a graphical method to analyse threats by decomposing an attacker's goal into subgoals. The root node denotes the attacker's objective, refined through conjunctive (all subgoals required) or disjunctive (one subgoal sufficient) refinements.

Attack trees are valuable in threat modelling and risk assessment, offering both intuitive visualisation and formal semantics that enable quantitative extensions (Mauw and Oostdijk, 2006; Kordy et al., 2014; Li et al., 2024). Yet, constructing them remains time-consuming, error-prone, and often subjective, especially as trees grow in size and complexity. To address this, research has focused on *attack-tree synthesis* (Audinot et al., 2017; Audinot et al., 2018; Gadyatskaya et al., 2017; Pinchinat et al., 2016; Vigo et al., 2014; Ivanova et al., 2015; Hong et al., 2013; Gadyatskaya, 2015; Wideł et al., 2019; Bryans et al., 2020; Pinchinat et al., 2020; Kern et al., 2021), i.e. automatically generating trees from system specifications. While mathematically rigorous, existing methods often sacrifice readability: some omit intermediate labels (Vigo et al., 2014) while others rely on templates or libraries (Bryans et al., 2020; Jhawar et al., 2018). However, no framework to date produces trees with formally defined labels while also optimising their structure. This article introduces the first such framework.

Contributions. We present an attack-tree synthesis framework that unifies and extends prior work by:

- Establishing a formal relation between system specifications, attacks, goals, and attack trees.
- Providing an interface for extensibility (Pinchinat et al., 2020; Gadyatskaya et al., 2017), allowing different system models to be plugged into the synthesis process.
- Using a formal language for labels (Mantel and Probst, 2019; Audinot and Pinchinat, 2016; Audinot et al., 2017; Audinot et al., 2018), ensuring unambiguous meaning and correctness with respect to the specification.
- Requiring attacks to explicitly state which goals they achieve, a feature only partially addressed in prior work.

Structure of the article. The remainder of this arti-

[a]This is a short version of an extended paper available at https://doi.org/10.48550/arXiv.2311.13331.

[b] https://orcid.org/0000-0002-3760-9165
[c] https://orcid.org/0000-0002-5843-0316
[d] https://orcid.org/0000-0002-8714-4626
[e] https://orcid.org/0000-0003-3049-7962

cle is organised as follows. Section 2 makes precise the problem statement we just introduced. Section 3 presents our solution approach via algebraic factorisation, which approximates the attack-tree minimisation problem. Sections 4 and 5 are devoted to labelling trees in a succinct and informative way: the former defines labels and their relation to attacks; the latter introduces an algorithm to label attack trees correctly. Conclusive remarks are given in Section 6.

2 Formulating the attack-tree synthesis problem

As we seek to build attack trees, the first step is to provide them with a formal syntax and semantics.

2.1 Attack tree syntax

Let $\mathbb{B}$ denote a set of *node labels*. Let OR be an unranked[1] associative and commutative operator, and SAND be an unranked associative but non-commutative operator. A SAND attack tree t is an expression over the signature $\mathbb{B} \cup \{\texttt{OR}, \texttt{SAND}\}$, generated by the following grammar (for $b \in \mathbb{B}$):

$$t ::= b \mid \texttt{OR}(t, \ldots, t) \mid \texttt{SAND}(t, \ldots, t).$$

The grammar defines attack trees with disjunctive and sequential-conjunctive operators. The OR operator specifies that achieving any child node suffices for the parent, while SAND requires all child nodes to be achieved in order. Readers familiar with attack trees may notice the absence of the AND operator in our grammar. We deliberately omit it from our syntax as our approach does not produce trees with AND nodes. In other words, all conjunctive refinements produced by our approach are sequential.

Next, we briefly introduce useful notations about the structure of attack trees. We write $\mathbb{T}_{\mathbb{B}}$ to denote all SAND attack trees. We say that t' is a *subtree* of $t = \Delta(t_1, \ldots, t_n)$, denoted $t' \preccurlyeq t$, if $t' = t$ or $t' \preccurlyeq t_i$ for some $i \in \{1, \ldots, n\}$, where $\Delta \in \{\texttt{OR}, \texttt{SAND}\}$. If t is a simple tree, i.e. a tree with a single node, then t itself is a subtree of t. When the subtree t' is not equal to t, we say that t' is a *proper subtree*. We use the auxiliary function $size(.)$ to denote the size of an attack tree measured in terms of its number of nodes. A recursive definition of the *size* function is as follows: $size(b) = 1$ if $b \in \mathbb{B}$; $size(\Delta(t_1, \ldots, t_n)) = size(t_1) + \cdots + size(t_n) + 1$ when $\Delta \in \{\texttt{OR}, \texttt{SAND}\}$.

[1] An unranked operator takes as argument an arbitrary number of terms.

2.2 Attack tree semantics

Because attack trees of different shape may well represent the same *attack scenario*, researchers have been providing attack trees with precise and unambiguous interpretations (Kordy et al., 2010; Jhawar et al., 2015). This makes it possible to guarantee consistency of analyses performed on attack trees and to determine whether two apparently different trees do cover the same attacks. In this article, we shall use a simplified version of the semantics (a.k.a. interpretation) of SAND attack trees given in (Jhawar et al., 2015); one that does not consider the AND operator.

Definition 2.1 (SAND tree semantics)**.** *Let $\mathbb{B}^+$ be all sequences formed by basic actions in $\mathbb{B}$ and $\cdot$ be the concatenation operator on sequences. The* semantics *for* SAND *attack trees is given by the function $[\![\cdot]\!] : \mathbb{T}_{\mathbb{B}} \to \mathcal{P}(\mathbb{B}^+)$, where $\mathcal{P}(\cdot)$ denotes the power-set, which is defined recursively as follows: for $b \in \mathbb{B}$, $t_i \in \mathbb{T}_{\mathbb{B}}$, $1 \leq i \leq k$,*

$$\begin{aligned} [\![b]\!] &= \{b\} \\ [\![\texttt{OR}(t_1, \ldots, t_k)]\!] &= \bigcup_{i=1}^{k} [\![t_i]\!] \\ [\![\texttt{SAND}(t_1, \ldots, t_k)]\!] &= \{a_1 \cdot \ldots \cdot a_k \mid \\ &\qquad a_1 \in [\![t_1]\!], \ldots, a_k \in [\![t_k]\!]\}. \end{aligned}$$

We use $t =_S t'$ to denote semantic equivalence of two trees $t, t' \in \mathbb{T}_{\mathbb{B}}$, which is defined by $t =_S t' \iff [\![t]\!] = [\![t']\!]$.

2.3 Attack-tree labelling

Even though the semantics of an attack tree is defined in terms of the labels of its leaf nodes, the labels of intermediate nodes are essential when using the tree for communication purposes. In this work, we follow the approach of (Mantel and Probst, 2019) and define a label as a propositional formula over basic actions. That is, basic actions are considered basic goals, and more complex goals are obtained by combining basic goals using propositional connectives.

Formally, a goal or label is a propositional formula over basic actions, denoted $\mathcal{F}(\mathbb{B})$. An attack-tree labelling is a partial function $\ell \colon \mathbb{T}_{\mathbb{B}} \hookrightarrow \mathcal{F}(\mathbb{B})$ that assigns a formula to attack trees with the following restriction: if t is a simple tree, i.e. $t \equiv b$ for some $b \in \mathbb{B}$, then $\ell(t) = b$. That is, the goal of a simple tree with basic action b is the basic goal b itself.

Definition 2.2 (Correctly-labelled tree)**.** *Let $\vdash$ be a relation from $\mathbb{B}^+$ to $\mathcal{F}(\mathbb{B})$ telling when an attack satisfies a goal. We say that a labelling function ℓ is a*

correct labelling of an attack tree t, with respect to $\vdash$, if every subtree t' in t satisfies that $\tau \in [\![t']\!] \implies \tau \vdash \ell(t')$.

In words, for a labelling function to be correct, it must be the case that every attack represented by a subtree t' of t satisfies the goal assigned to t'.

It is worth highlighting that our correct-labelling is stronger than that of (Mantel and Probst, 2019), where only the root node is required to be correctly labelled. We argue, however, that every node in the tree should be correctly labelled, as it provides a top-down interpretation of the attack scenario.

2.4 Problem statement

We are finally in a position to state the problem we address in this article.

Definition 2.3 (The attack-tree synthesis problem)**.** *Let $\mathcal{A} \subseteq \mathbb{B}^+$ be a set of attacks and $\vdash \subseteq \mathbb{B}^+ \times \mathcal{F}(\mathbb{B})$ a relation that specifies which goals (basic or complex) are satisfied by a given attack.* The attack-tree synthesis problem *consists in finding a* SAND *tree t together with a correct-labelling function ℓ such that the tree is sound, i.e. $[\![t]\!] = \mathcal{A}$, and of minimum-size, i.e. $\exists t' \colon size(t') < size(t) \implies [\![t']\!] \neq [\![t]\!]$.*

It is worth remarking that, while most previous works address soundness, they often overlook the properties of minimum-size and correct-labelling. We do not claim those properties are strictly necessary, nor that they lead to the best trees. We argue, however, that the absence of any structural and labeling criteria results in attack trees of arbitrary shape and labels, thereby making the analysis of their merits a purely subjective task.

3 Attack trees of minimum size via algebraic factorisation

To generate attack trees of minimum size, we resort to algebraic factorization, which has been long studied in algebra for the minimisation of digital circuits. In this case, we use it for the minimisation of attack trees.

Definition 3.1 (Reducibility and the factored form for attack trees)**.** *We say that a set of attacks $\mathcal{A}$ is* reducible *iff there exists a* SAND *tree of the form $t = b \triangleleft \mathtt{SAND}(t_1, \ldots, t_n)$ such that $\mathcal{A} = [\![t]\!]$. Moreover, we say that t is a* factorisation *of $\mathcal{A}$ and that t is in the* factored form *if $[\![t_1]\!]$, ..., $[\![t_n]\!]$ are irreducible.*

Consider the following set of attacks $\{a \cdot b, c \cdot d\}$. In this case, it is easy to prove that an attack tree with minimum size with such semantics is $t = \mathtt{OR}(\mathtt{SAND}(a,b), \mathtt{SAND}(c,d))$. The proof follows from the observation that $\{a \cdot b, c \cdot d\}$ is *irreducible*, in the sense that it cannot be expressed by an attack tree of the form $\mathtt{SAND}(t_1, \ldots, t_n)$.

Our heuristics to synthesise attack trees of minimum size thus consists of creating attack trees in the factored form whenever possible. And, we do so by reducing the attack-tree factorisation problem into the algebraic factorisation problem; the latter has been studied for decades due to its applications to the optimisation of digital circuits. Our reduction works thanks to the following algebraic property of the semantical domain of our SAND trees.

Lemma 3.1. *The algebraic structure $(\mathcal{P}(\mathbb{B}^+), \cup, \cdot)$, where $\cup$ is the standard set-join operator and $\cdot$ is the concatenation operator generalised on sets and defined by $\forall A, B \in \mathcal{P}(\mathbb{B}^+) \colon A \cdot B = \{a \cdot b | a \in A, b \in B\}$, is an idempotent semiring with no multiplicative identity.*

Proof. All proofs can be found in the extended version of the paper at https://doi.org/10.48550/arXiv.2311.13331. □

The above lemma is relevant because, should an algebraic factorisation procedure for idempotent semirings exist, we could directly use such procedure to factorise attack trees, as we show next.

Lemma 3.2. *Let S be an element of $(\mathcal{P}(\mathbb{B}^+), \cup, \cdot)$. Let $S_1 \cdot \ldots \cdot S_n \cup R$ be a factorisation of S with $n \geq 1$. For every $i \in \{1, \ldots, n\}$, let t_i be a* SAND *tree in factored form with semantics S_i. If $n > 1$, then the tree $\mathtt{SAND}(t_1, \ldots, t_n)$ is a* SAND *tree in factored form with semantics $S_1 \cdot \ldots \cdot S_n$. If $n = 1$, then there does not exist a* SAND *tree in factored form with semantics S_1.*

A corollary of the lemma above is twofold. If $R = \emptyset$, then the tree $\mathtt{SAND}(t_1, \ldots, t_n)$ is a SAND tree in factored form with semantics S, which is what we want to obtain. If $R \neq \emptyset$, then a tree in factored form with semantics S cannot be built. In that case, however, we can still build a tree with semantics S consisting of a root node that disjunctively branches into two trees t_1 and t_2 such that t_1 is in factored form. The tree t_1 is defined by $\mathtt{SAND}(t_1, \ldots, t_n)$, and t_2 being a SAND tree with semantics R. Soon we will give a complete algorithmic description of our attack-tree synthesis approach via algebraic factorisation. But, first, we shall address the algebraic factorisation problem.

3.1 A factorisation heuristic for expressions in idempotent semirings

Even though algebraic factorisation has been extensively studied and continues to be an active research area in algebra and number theory, we are not aware of a factorisation procedure for idempotent semirings. On the one hand, there exist many factorisation techniques for polynomials over finite fields, but they are not idempotent. On the other hand, many works have addressed the minimisation of circuit design (Brayton et al., 1987) via factorisation of logic functions, which are idempotent. However, multiplication of logic functions is commutative, while our multiplication (concatenation) is not. This leaves, to the best of our knowledge, the problem of factorising expressions in $(\mathcal{P}(\mathbb{B}^+),\cup,\cdot)$ open, which we address next.

For simplicity, given that the reader might be familiar with standard algebraic manipulations, we use $x_1\cdots x_n$ as a shorthand notation for $\{x_1\cdot\ldots\cdot x_n\}\in\mathcal{P}(\mathbb{B}^+)$, x^n instead of $\underbrace{x\cdots x}_{n}$, and $a+b$ as a shorthand notation for $a\cup b$ with $a,b\in\mathcal{P}(\mathbb{B}^+)$. This gives, for example, a^2+ab+b^2 instead of $\{\{a\cdot a\},\{a\cdot b\},\{b\cdot b\}\}$. We remark, nonetheless, that the product operator here should not be assumed to be commutative. For example, aba is not equal to a^2b, unless $\cdot$ is commutative.

The solution we propose is inspired by the factorisation procedure for logic functions introduced in (Brayton et al., 1987). Let us start by introducing the necessary notations.

- A *cube* is a set $\{x_1\cdot\ldots\cdot x_n\}\in\mathcal{P}(\mathbb{B}^+)$ of cardinality one denoting the product $x_1\cdots x_n$. It is the simplest algebraic expression one can form, such as b^2 and ab with $a,b\in\mathbb{B}$.
- An expression in the Sum of Product form (SoP) is an expression of the form $c_1+\ldots+c_n$ where $c_1,\ldots,c_n$ are cubes.
- The *length* of an expression f is the minimum positive integer n such that there exists an SoP expression $c_1+\ldots+c_n$ equal to f.
- x is said to be a left-divisor of f if y is the largest expression, called the quotient, such that $f=x\cdot y+r$ for some r called the remainder of the division. For example, a is a left-divisor of a^2+ab+b^2 with quotient $a+b$ and remainder b^2. Notice that b is not a left-divisor of a^2+ab+b^2, unless $\cdot$ was commutative.
- x is said to be a right-divisor of f if y is the largest expression, called the quotient, such that $f=y\cdot x+r$ for some r called the remainder of the division. For example, b is a right-divisor of a^2+ab+b^2 with quotient $a+b$ and remainder a^2. Notice that a is not a right-divisor of a^2+ab+b^2, unless $\cdot$ was commutative.

Proposition 3.1. *Let $f=c_1+\cdots+c_m$ be an expression in SoP. Let R_f be the set of all pairs (x,y) such that $x\cdot y=c_i$ for some $i\in\{1,\ldots,m\}$. Let $\pi_1(R_f,z)=\{y|(z,y)\in R_f\}$ and $\pi_2(R_f,z)=\{x|(x,z)\in R_f\}$ be the projections of the first and second component of R_f, respectively, onto z. If $\{x_1,\ldots,x_n\}$ is a set of maximum cardinality such that $\pi_1(R_f,x_1)\cap\cdots\cap\pi_1(R_f,x_n)$ is non-empty, then $x_1+\cdots+x_n$ is the quotient of the right-division of f by $\pi_1(R_f,x_1)\cap\cdots\cap\pi_1(R_f,x_n)$. Likewise, if $\{y_1,\ldots,y_n\}$ is a set of maximum cardinality such that $\pi_2(R_f,y_1)\cap\cdots\cap\pi_2(R_f,y_n)$ is non-empty, then $y_1+\cdots+y_n$ is the quotient of the left-division of f by $\pi_2(R_f,y_1)\cap\cdots\cap\pi_2(R_f,y_n)$.*

Based on the proposition above, we build a heuristic (described in Algorithm 1) to factorise expressions in $(\mathcal{P}(\mathbb{B}^+),\cup,\cdot)$ by greedily looking for a quotient x and a divisor y such that the product of their length is maximum.

A note on the complexity of the factorisation algorithm. The complexity of ExpFactorisation is dominated by the while loop at line 5. In the worst case, the loop iterates $|R|$ times. Each iteration requires, in the worst case, computing the projections π_1 and π_2 for all elements in R. Because calculating π_1 and π_2 is linear in $|R|$, we obtain an overall complexity of $O(|R|^3)$ for ExpFactorisation. Now, in the worst case, R contains all possible pairs of prefixes and suffixes of the cubes in f. Therefore, $|R|$ is $O(nm)$ where n is the maximum length of the cubes in f and m is the number of cubes in f. Hence, the overall complexity of ExpFactorisation is $O(n^3m^3)$.

3.2 A factorisation heuristic for attack trees

We conclude this section by showing how the previous factorisation algorithm can be applied to approximate the problem of generating an attack tree of minimum size with a given semantics. The approach goes as follows, given a set of attacks $\mathcal{A}$:

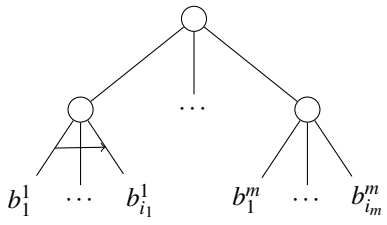
1. Transform $\mathcal{A}$ into an algebraic expression in SoP form. This is done in the straightforward way: each attack $b_1\cdot\ldots\cdot b_n\in\mathcal{A}$ is a *cube* and, therefore, $\mathcal{A}$ is already an expression in the SoP form. In other words, $\mathcal{A}$ is an element of the idempotent semiring $(\mathcal{P}(\mathbb{B}^+),\cup,\cdot)$.
2. If $\mathcal{A}$ contains a single attack $b_1\cdot\ldots\cdot b_m$, then the following attack tree has minimum size: $\texttt{SAND}(b_1,\ldots,b_n)$.

Algorithm 1: EXPFACTORISATION(f)

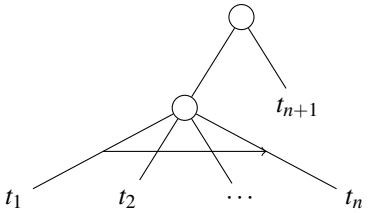
Input: An expression f in SoP over an idempotent semiring $(\mathcal{P}(\mathbb{B}^+), \cup, \cdot)$
Output: A factorisation of f of the form $g_1 \cdots g_n + r$ where $g_1, \ldots, g_n$ are irreducible

Let $R \leftarrow$ all pairs (x,y) such that $x \cdot y = c$ with c a cube in f;
if *R is empty* **then**
  return f
Let $X \leftarrow \emptyset$, $Y \leftarrow \emptyset$;
while *R is not empty* **do**
  Let $x_{\max}$ such that $\pi_1(R, x_{\max})$ has maximum cardinality;
  Let $y_{\max}$ such that $\pi_2(R, y_{\max})$ has maximum cardinality;
  if $|x_{max}| \geq |y_{max}|$ **then**
    Add $x_{\max}$ to X;
    $Y \leftarrow \pi_1(R, x_{\max})$;
    Remove all pairs (x,y) from R such that $y \notin \pi_1(R, x_{\max})$ or $x = x_{\max}$;
  else
    Add $y_{\max}$ to Y;
    $X \leftarrow \pi_2(R, y_{\max})$;
    Remove all pairs (x,y) from R such that $x \notin \pi_2(R, y_{\max})$ or $y = y_{\max}$;
if $X = \emptyset$ **then**
  return f ; `// No further factorisation is possible`
Let $X = \{x_1, \ldots, x_n\}$ and $x = x_1 + \cdots + x_n$;
Let $Y = \{y_1, \ldots, y_m\}$ and $y = y_1 + \cdots + y_m$;
Let r such that $f = x \cdot y + r$;
Let $x' \leftarrow$ EXPFACTORISATION(x);
Let $y' \leftarrow$ EXPFACTORISATION(y);
return $x' \cdot y' + r$

3. If $\mathcal{A}$ contains multiple attacks, factorize $\mathcal{A}$ by calling Algorithm 1, obtaining a factorized form $q_1 \cdot \ldots \cdot q_n \cup r$.
4. If $\mathcal{A}$ is irreducible, i.e. if $n = 1$, then produce an attack tree of the following form, with $\mathcal{A} = \{b_1^1 \cdots b_{i_1}^1, \ldots, b_1^m \cdots b_{i_m}^m\}$:



5. If $n > 1$, let t_i be an attack tree whose semantics is q_i, for every $i \in \{1, \ldots, n\}$, and t_{n+1} be an attack tree whose semantics is r. It follows that the following attack tree has semantics exactly $\mathcal{A}$.



Algorithm 2 consolidates and make precise the five steps described above into a pseudocode specification of our attack-tree minimisation method.

Algorithm 2: TREEMINIMIZATION($\mathcal{A}$)

Input: An non-empty set of attacks $\mathcal{A}$
Output: A tree t with semantics $\mathcal{A}$, optimised for minimum size via factorisation

if $\mathcal{A} = \{\tau\}$ **then**
  Let $\tau = b_1 \cdot \ldots \cdot b_n$;
  if $n = 1$ **then**
    return b_1 ; `// A tree with a single node` b_1
  else
    return SAND$(b_1, \ldots, b_n)$;
Factorise $\mathcal{A}$ into $\mathcal{A}_1 \cdot \ldots \cdot \mathcal{A}_n \cup \mathcal{A}_{n+1}$ using EXPFACTORISATION (Algorithm 1) ; `//` $\mathcal{A}_{n+1}$ `is the remainder`
if $n = 0$ **then**
  foreach $\tau_i \in \mathcal{A}$ **do**
    $t_i \leftarrow$ TREEMINIMIZATION($\{\tau_i\}$);
  return OR$(t_1, \ldots, t_m)$; `// Flat tree`
else if $\mathcal{A}_{n+1} = \emptyset$ **then**
  foreach $\mathcal{A}_i$ **do**
    $t_i \leftarrow$ TREEMINIMIZATION($\mathcal{A}_i$);
  return SAND$(t_1, \ldots, t_n)$;
else
  $t_l \leftarrow$ TREEMINIMIZATION($\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$);
  $t_r \leftarrow$ TREEMINIMIZATION($\mathcal{A}_{n+1}$);
  return OR(t_l, t_r);

A note on the complexity of the tree-minization algorithm. The complexity of TREEMINIMISATION is equal to the complexity of EXPFACTORISATION. That is, TREEMINIMISATION has complexity $O(n^3 m^3)$ where m is the size of the semantics of input tree t and n is the maximum length of the attacks in the semantics of t. This is because, although the algorithm makes recursive calls it itself, each call works

on a member of a partition of the semantics of t. In this case, the cubic complexity of TREEMINIMISATION dominates over the complexity of the recursive calls.

3.3 Correctness analysis

As far as correctness goes, all we need to show is that Algorithm 2 produces sound trees, as we have already established that we approximate the tree size via factorisation.

Theorem 3.1. *Algorithm 2 produces sound trees.*

Proof. We prove soundess inductively. At the base case we have $\mathcal{A} = \{\tau\}$ with $\tau = b_1 \cdot ... \cdot b_n$. This produces the tree $\texttt{SAND}(b_1, ..., b_n)$, whose semantics is effectively $\{b_1 \cdot ... \cdot b_n\}$; hence the tree is sound.

For the inductive step, assume that for every set of attacks $\mathcal{A}$ with $|\mathcal{A}| \leq k$, Algorithm 2 produces sound trees. Consider $\mathcal{A}$ with $|\mathcal{A}| = k+1$. Three cases arise:
Case 1: $\mathcal{A}$ irreducible. The factorisation returns $\mathcal{A}_1 = \mathcal{A}$. For each $\tau_i \in \mathcal{A}$, the algorithm recursively builds a sound tree t_i, and outputs $\texttt{OR}(t_1, ..., t_m)$ with semantics $\{\tau_1, ..., \tau_m\} = \mathcal{A}$.
Case 2: $\mathcal{A}$ factorisable. Let $\mathcal{A} = \mathcal{A}_1 \cdot \cdots \cdot \mathcal{A}_n$. For each $\mathcal{A}_i$, the algorithm builds a sound tree t_i by induction. The output $\texttt{SAND}(t_1, ..., t_n)$ has semantics $\mathcal{A}_1 \cdot \cdots \cdot \mathcal{A}_n = \mathcal{A}$.
Case 3: $\mathcal{A}$ reducible but not factorisable. Here $\mathcal{A} = \mathcal{A}_1 \cdot \cdots \cdot \mathcal{A}_n \cup \mathcal{A}_{n+1}$ with $\mathcal{A}_{n+1} \neq \emptyset$. The algorithm builds t_l and t_r, sound by induction for the two parts, and outputs $\texttt{OR}(t_l, t_r)$ with semantics $\mathcal{A}$.

□

4 A system model to express attacks and their goals

Now that we have presented an algorithm to build sound attack trees of minimum size, we need to address the question of how to label the nodes of the tree. This requires a concrete interpretation of a relation $\vdash$ between attacks and labels, and a strategy to label the tree in a correct way with respect to $\vdash$. This section addresses the first challenge, while the next section introduces a labelling procedure for attacks trees.

4.1 Mixed Kripke Structure

Our underlying execution model is a variation of the *Labelled Transition Systems* (LTS), called a Kripke structure, which are used to describe the behaviour of a system by defining the transitions that bring a system from one state into another. Each state in a Kripke structure contains properties that hold in the corresponding state, such as the knowledge of an attacker or the location of a secret document. The transition system progresses via application of transition rules until a final state is reached.

Formally, a *Mixed Kripke Structure* is a quadruple $(\mathcal{S}, \Lambda, \rightarrow, s_0, F)$, where $\mathcal{S}$ is a set of *states*; Λ is a set of *labels*; $\rightarrow \subseteq \mathcal{S} \times \Lambda \times \mathcal{S}$ is a *transition relation*; $s_0 \in \mathcal{S}$ is the *initial state* and $F \subset \mathcal{S}$ is a set of final states. We usually write $s \xrightarrow{a} s'$ to say that $(s, a, s') \in \rightarrow$. And we say that $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \cdots s_{n-1} \xrightarrow{a_{n-1}} s_n$ is a path if $s_n \in F$ and $s_i \xrightarrow{a_i} s_{i+1}$ for every $i \in \{0, ..., n-1\}$. Any portion of the path formed by one or more consecutive transitions is called a *subpath*. When s_0 is a final state, then s_0 itself is a path with no transition. We use $paths(l)$ to denote the set of all paths generated by a labelled transition system l.

States. As mentioned earlier, states contain system properties, which we formalise as predicates constructed over a signature Σ (i.e. a collection of function symbols). Function symbols of arity zero are called constants and are written without parentheses, such as the name of a system actor (e.g. *Bob*) or the name of an item (e.g. *pwd* to refer to a password). Function symbols with non-zero arity surround within parentheses their arguments, such as $\texttt{knw}(Bob, pwd)$ to denote that *Bob* knows password *pwd*.

The transition relation. The transition relation is defined through *transition rules*. Every transition rule is of the form $[name]\frac{C}{s \xrightarrow{a} f(s)}$, and contains a condition C and a conclusion $s \xrightarrow{a} f(s)$. The name of a transition rule is given left of the line. The condition C consists of zero or more predicates p that must be present in state s to enable the transition rule. The conclusion $s \xrightarrow{a} f(s)$ describes the state change when the transition occurs, with f a function mapping old state s to a new state $f(s)$. The arrow is labelled with the event that describes the transition. The predicates may contain variables which are implicitly universally quantified.

4.2 Goals and attacks

We shall define an *attack* as a sequence of *attack goals*. Because an execution of the system is a sequence of transitions of the type $s \xrightarrow{a} s'$ where the action a leads the system into the state s' by removing and adding system properties from s, we consider the *goal* of such transition to be exactly those properties that have been added/removed from s. Like-

wise, the goal of a sequence of transitions $s_i \xrightarrow{a_i} s_{i+1} \cdots s_{j-1} \xrightarrow{a_{j-1}} s_j$ consists of adding to and removing from s_i properties in such a way that the system reaches state s_j. This leads to the following definition of attacks, goals, and their relation.

Definition 4.1 (Goals)**.** *Given an LTS l, an* attack goal *is any pair of sets of predicates $(P^-, P^+) \in \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P})$. We say that an LTS sub-path path $s_i \xrightarrow{a_i} s_{i+1} \cdots s_{j-1} \xrightarrow{a_{j-1}} s_j$* satisfies *the goal (P^-, P^+) if the following two conditions hold:*

1. *$P^- \subseteq s_i \setminus s_j$, meaning that each property in P^- is part of the starting state s_i but not of the last state s_j.*
2. *$P^+ \subseteq s_j \setminus s_i$, meaning that each property in P^+ is part of the last state s_j but not of the starting state s_i.*

We use the relation $\vdash$ to denote that a subpath satisfies a goal.

Now that we defined what constitutes an attack goal, we proceed to define an attack as a sequence of goals.

Definition 4.2 (Attack)**.** *Given an LTS l, we say that a sequence of goals $G = (P^-_{i_1}, P^+_{i_1}) \cdot \ldots \cdot (P^-_{i_n}, P^+_{i_n})$ is an* attack *if there exists an LTS subpath $s_{i_1} \xrightarrow{a_{i_1}} s_{i_2} \xrightarrow{a_{i_2}} \cdots \xrightarrow{a_{i_{n-1}}} s_{i_n} \xrightarrow{a_{i_n}} s_{i_{n+1}}$ such that, for every $j \in \{1, \ldots, n\}$ $s_{i_j} \xrightarrow{a_{i_j}} s_{i_{j+1}} \vdash (P^-_{i_j}, P^+_{i_j})$, i.e. each transition of the LTS subpath satisfies the corresponding goal of the sequence.*

5 Tree labelling

In this last section we show how to label attack trees whose semantics is given in terms of the system model introduced earlier. Like with the size of the trees, we aim at producing succinct yet informative labels. On that direction, our first step is showing how to derive a label for a set of attacks.

Proposition 5.1. *Let $\mathcal{A} = \{\tau_1, \ldots, \tau_n\}$ be set of attacks as defined in Section 4. For each $\tau_j = b^j_1 \cdots b^j_{i_j}$, let $s^j_1 \xrightarrow{a^j_1} s^j_2 \cdots s^j_{i_j} \xrightarrow{a^j_{i_j}} s^j_{i_j+1}$ be the LTS subpath satisfying, for every $k \in \{1, \ldots, i_j\}$, $b^j_k = (s^j_k \setminus s^j_{k+1}, s^j_{k+1} \setminus s^j_k)$. Consider now the goal b defined by $b = ((s^1_1 \setminus s^1_{i_1+1}) \cap \ldots \cap (s^n_1 \setminus s^n_{i_n+1}), (s^1_{i_1+1} \setminus s^1_1) \cap \ldots \cap (s^n_{i_n+1} \setminus s^n_1))$. It follows by definition that $\tau_j \vdash b$ for every $j \in \{1, \ldots, n\}$.*

Our labelling strategy (described in Algorithm 3) for attack trees consists of producing labels by exploiting Proposition 5.1. This has the advantage of producing labels that are both informative and succinct. They are succinct because they consists of a basic goal. They are informative because they contain all properties that are common to all attacks in the set. In case the label produced by Proposition 5.1 is $(\emptyset, \emptyset)$, which is the weakest possible label, we label the node with the disjunction of its children labels. This ensures that every node in the tree is labelled with a goal that is satisfied by all attacks in the set.

Algorithm 3: TREELABELLING(t)

Input: An attack tree t
Output: A correctly-labelling ℓ for t

if $t \equiv b$ **then**
  return the labelling $\{(t, b)\}$;
if $t \equiv \Delta(t_1, \ldots, t_n)$ *with* $\Delta \in \{\texttt{OR}, \texttt{SAND}\}$ **then**
  $\forall i \in \{1, \ldots, n\}$, $\ell_i \leftarrow$ TREELABELLING(t_i) ; `// Recursive call`
  Let b be the label obtained by applying Proposition 5.1 on input $[\![t]\!]$;
  if $b \neq (\emptyset, \emptyset)$ **then**
    return $\ell_1 \cup \cdots \cup \ell_n \cup \{(t, b)\}$
  else
    return $\ell_1 \cup \cdots \cup \ell_n \cup \{(t, \ell_1(t_1) \vee \cdots \vee \ell_n(t_n))\}$; `// t is labelled with the disjunction of its children labels`

6 Conclusions

We have introduced the first attack-tree synthesis framework that optimises the shape of the generated trees while also producing short yet informative labels. Particularly relevant is our reduction of the tree minimisation problem to the factorisation problem of algebraic expressions, which calls for further research on the factorisation of (commutative) semirings without multiplicative identity. We showed the feasibility of our approach by introducing a system model that can relate attacks and goals. Future work will be directed towards interfacing the generation framework with a model checker tool, such as mCRL2[2], to allow for a large-scale validation of the approach. Such validation may include a study of whether the objective functions of our attack-tree generation problem, namely minimisation of size, do lead to trees that are easier to read and comprehend. To translate labels specified within a formal language into human-

[2] https://www.mcrl2.org/web/index.html

readable labels, we also plan to investigate natural language generation techniques.

Acknowledgements

Rolando Trujillo Rasua was supported by a Ramón y Cajal grant (RYC2020-028954-I) from the Spanish Ministry of Science and Innovation and the EU, as well as by projects PROVTOPIA (PID2023-150098OB-I00), funded by MICIU/AEI/10.13039/501100011033 and FEDER (EU) and HERMES, funded by INCIBE and the EU's NextGenerationEU/PRTR.

REFERENCES

Audinot, M. and Pinchinat, S. (2016). On the soundness of attack trees. In Kordy, B., Ekstedt, M., and Kim, D. S., editors, *Graphical Models for Security*, pages 25–38, Cham. Springer International Publishing.

Audinot, M., Pinchinat, S., and Kordy, B. (2017). Is my attack tree correct? In *European Symposium on Research in Computer Security*, pages 83–102. Springer.

Audinot, M., Pinchinat, S., and Kordy, B. (2018). Guided design of attack trees: a system-based approach. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 61–75. IEEE.

Brayton, R., Rudell, R., Sangiovanni-Vincentelli, A., and Wang, A. (1987). Mis: A multiple-level logic optimization system. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 6(6):1062–1081.

Bryans, J., Liew, L. S., Nguyen, H. N., Sabaliauskaite, G., Shaikh, S., and Zhou, F. (2020). A template-based method for the generation of attack trees. In Laurent, M. and Giannetsos, T., editors, *Information Security Theory and Practice*, pages 155–165, Cham. Springer International Publishing.

Gadyatskaya, O. (2015). How to generate security cameras: Towards defence generation for socio-technical systems. In *Proc. of GraMSec*. Springer.

Gadyatskaya, O., Jhawar, R., Mauw, S., Trujillo-Rasua, R., and Willemse, T. A. C. (2017). Refinement-aware generation of attack trees. In *Security and Trust Management - 13th International Workshop, STM 2017, Oslo, Norway, September 14-15, 2017, Proceedings*, volume 10547 of *LNCS*, pages 164–179.

Hong, J. B., Kim, D. S., and Takaoka, T. (2013). Scalable attack representation model using logic reduction techniques. In *Proc. of TrustCom*. IEEE.

Ivanova, M. G., Probst, C. W., Hansen, R. R., and Kammueller, F. (2015). Transforming graphical system models to graphical attack models. In *Proc. of GraMSec*. Springer.

Jhawar, R., Kordy, B., Mauw, S., Radomirović, S., and Trujillo-Rasua, R. (2015). Attack trees with sequential conjunction. In *Proc. of SEC*, volume 455 of *IFIP AICT*. Springer.

Jhawar, R., Lounis, K., Mauw, S., and Ramírez-Cruz, Y. (2018). Semi-automatically augmenting attack trees using an annotated attack tree library. In Katsikas, S. K. and Alcaraz, C., editors, *Security and Trust Management*, pages 85–101, Cham. Springer International Publishing.

Kern, M., Liu, B., Betancourt, V. P., and Becker, J. (2021). Model-based attack tree generation for cybersecurity risk-assessments in automotive. In *2021 IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–7.

Kordy, B., Mauw, S., Radomirović, S., and Schweitzer, P. (2010). Foundations of Attack–Defense Trees. In Degano, P., Etalle, S., and Guttman, J. D., editors, *FAST*, volume 6561 of *LNCS*, pages 80–95. Springer.

Kordy, B., Mauw, S., Radomirovic, S., and Schweitzer, P. (2014). Attack-defense trees. *Oxford Univ. Press J. Logic and Computation*, 24(1):55–87.

Li, S., Wu, S., and Guo, X. (2024). Security risk assessment method for measurement control systems based on attack tree model and common vulnerability scoring system. In *Proceedings of the 2023 International Conference on Artificial Intelligence, Systems and Network Security*, AISNS '23, page 360–369, New York, NY, USA. Association for Computing Machinery.

Mantel, H. and Probst, C. W. (2019). On the meaning and purpose of attack trees. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 184–18415.

Mauw, S. and Oostdijk, M. (2006). Foundations of attack trees. In Won, D. H. and Kim, S., editors, *Information Security and Cryptology - ICISC 2005*, pages 186–198, Berlin, Heidelberg. Springer Berlin Heidelberg.

Pinchinat, S., Acher, M., and Vojtisek, D. (2016). Atsyra: An integrated environment for synthesizing attack trees. In Mauw, S., Kordy, B., and Jajodia, S., editors, *Graphical Models for Security*, pages 97–101, Cham. Springer International Publishing.

Pinchinat, S., Schwarzentruber, F., and Lê Cong, S. (2020). Library-based attack tree synthesis. In *International Workshop on Graphical Models for Security (GraMSec)*, pages 24–44. Springer.

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

Vigo, R., Nielsen, F., and Nielson, H. R. (2014). Automated generation of attack trees. In *Proc. of CSF*, pages 337–350. IEEE.

Wideł, W., Audinot, M., Fila, B., and Pinchinat, S. (2019). Beyond 2014: Formal methods for attack tree–based security modeling. *ACM Computing Surveys (CSUR)*, 52(4):1–36.