Attack trees: a notion of missing attacks

Florence Wacheux[1], Sophie Pinchinat[1], Barbara Fila[2], and Yann Thierry-Mieg[3]

[1] Univ Rennes, CNRS, IRISA, Rennes, France
[2] Univ Rennes, INSA Rennes, CNRS, IRISA, Rennes, France
[3] Sorbonne Université, CNRS, LIP6, Paris, France

Abstract. Attack trees are widely used for security modeling and risk analysis. Classically, an attack tree combines possible actions of the attacker into attacks. In most existing approaches, an attack tree represents generic ways of attacking a system, but without taking any specific system or its configuration into account. This means that such a generic attack tree may contain attacks that are not applicable to the analyzed system, and also that a given system could enable some attacks that the attack tree did not capture.
To overcome this problem, we extend the attack tree setting with a model of the analyzed system, allowing us to introduce precise *path semantics* of an attack tree and to define *missing attacks*. We investigate the *missing attack existence problem* and show how to solve it by calls to the NP oracle that answers the *trace attack tree membership problem*; the latter problem has been implemented and is available as an open source prototype.

Keywords: Risk analysis, attack trees, path semantics, missing attacks, complexity

1 Introduction

Attack trees are well-known graphical models for security analysis. They were introduced by Schneier twenty years ago [23] and since then they have been successfully adopted by industry [7,19,1,11,8], as well as gained a lot of popularity within the scientific security community [15,10]. The hierarchical structure of an attack tree refines the goal of an attacker (depicted by the root node) into simpler sub-goals, using disjunctive and conjunctive nodes. At the bottom of the tree we find the leaves that correspond to actions that the attacker needs to perform to reach his goal. Combinations of such actions form attacks that lead to achieving the root goal of the tree. Numerous formalizations of attacks in the context of attack trees have been proposed: multisets [18], sets [16], models of Boolean functions [14], special types of graphs [13], paths or traces [3,2], etc. In this paper, we model possible attacks as sequences of the attacker's actions.

A lot of research effort has recently been put into the problem of generation of attack trees [25,9,21,12,3]. Regardless of whether such generation is manual or automated, two main approaches can be distinguished: a *generic approach*, where the constructed attack tree covers a large number of well-known, generic attacks, applicable to most potential attackers; and a *specific approach*, where an attack tree is tailored to a specific system and/or to a specific attacker profile. In the first case, the creation process may benefit from the usage of attack tree libraries or commonly-known attack patterns, such as [19], whereas in the second case, a suitable attack tree is usually derived from a formal model of the analyzed system, e.g., [12,20,21]. Unfortunately, both approaches have their limitations. On the one hand, attack trees created using the generic approach may contain attacks that are irrelevant for the analysis of a given system – that we call *extra attacks*. On the other hand, in the case of the specific approach, the modeler can easily miss some attacks, for instance because he is not aware of particularities of the system. In this case, we talk about *missing attacks*. In both – generic and specific – approaches, it is also possible that the tree contains some weird attacks or misses some other ones due to an inappropriate expertise of the modeler, or because the formal model of the analyzed system is too coarse or too abstract.

We argue that, in order to perform a decent security analysis, an attack tree model needs to be coupled with the formal model of the analyzed system. Indeed, the former represents how the system can be attacked, whereas the latter describes how this system actually looks like. Taking both models into account simultaneously provides an elegant way of formally verifying the relevance of an attack tree w.r.t. the system, in terms of extra and missing attacks. The presence of the system model also allows us to extend the attack tree formalism with *weak* refinement operators that are used to refine goals in a more flexible manner. The specific contributions of this work are the following:

- We accompany an attack tree with an explicit modeling of the analyzed system, using a labeled transition system, which allows us to propose a new semantics for attack trees – *path semantics* – formalizing attacks in terms of sequences of attacker's actions corresponding to paths in the analyzed system. The use of this model of the system automatically discards extra attacks in the semantics.
- We extend `OR-AND-SAND` attack trees with the weak conjunctive (`wAND`) and weak sequential (`wSAND`) operators, allowing an expert to model collections of actions that are necessary but might not be sufficient for the attacker to reach his goal.
- We formally define two decision problems: *trace attack tree membership* (TATM) and *missing attack existence* (MAE). The first one focuses on whether a given attack is covered by an attack tree; the second, whether the tree contains any missing attacks w.r.t. the considered system.
- We provide algorithms solving the two problems. We prove that TATM is NP-complete, and that MAE for trees with no weak operators is in the second level of the polynomial hierarchy [24] resorting to the NP oracle for TATM.

This paper is structured as follows. Section 2 describes the relevant existing work. In Section 3, we present the background knowledge on attack trees that is necessary to understand the framework proposed in this article. We extend the attack tree model with a formalization of the system under study in Section 4. We introduce the concept of missing attacks, study the missing attack existence decision problem, and keep on with the trace membership problem in Section 5. In this section we also briefly describe the tool that we implemented to automate the solving of the TATM problem. We conclude and discuss future research directions in Section 6.

2 Related work

Although the attack tree literature is very abundant [15,10], only two lines of research involve an explicit modeling of the analyzed system.

The first direction of work combining attack trees and a system model concentrates on attack tree correctness. In [2] and [4], a transition system whose states are labeled with propositions that express possible configurations of the underlying real-life system is employed. The authors define a novel kind of attack trees, called *state-based attack trees*, where the attacker's goals, i.e., nodes' labels, are expressed with two propositions representing the initial configuration, from which the attacker starts his attack, and a final configuration that the attacker aims at. Since the system model and the attack tree use a common language of propositions, it is possible to identify the set of paths in the system that allow the attacker to reach the goal of a specific node, i.e., to go from its initial to its final configuration. By using appropriate combination operators, similar to the sequential and parallel composition of paths used it this work, one can thus check whether an attack tree represents at least one valid attack in the analyzed system (non-emptiness problem [4]), as well as verify the quality of a node's refinement w.r.t. the system [2], i.e., check whether all paths satisfying the goal of the parent node also satisfy the combination of the goals of its children (over-match), and vice-versa (under-match).

The concept of path semantics used in the current work is very similar to the one from [2] and [4]. The main difference consists in the way in which attack tree node labels are formalized: goals over propositions, in the state-based approach, versus attacker's actions and their combinations, in the present work.

Supporting attack tree generation is the second research direction that involves reasoning about a particular system in the context of attack trees. The objective of [12] is to create an attack tree describing how a given socio-technical system, e.g., a company, can be attacked. The system is represented with a graph-based model capturing its locations, actors and processes involved, and relevant assets. Conditions, called policies, define which actions can be performed by which actor or process, and how. An attacker's goal is expressed in terms of policy invalidation: actors, assets, and locations necessary to enact a policy are determined, and the corresponding path in the system model is identified. An algorithm recursively constructs an attack tree for every policy, and then combines them using an `AND` node to get a tree representing a single path. Finally, the trees corresponding to particular paths are combined under a common `OR` node, resulting in a tree invalidating the initial policy.

In [20] and [21], the authors address the problem of generating an attack tree for a physical system formalized using a domain specific language. The system description is compiled into a symbolic transition system, and the reachability analysis, based on model checking, is performed to generate attack scenarios expressed as sequences of elementary actions of the attacker. By combining these conjunctive scenarios using an `OR` node, a flat attack tree is obtained. Parsing and merging are then used to factorize the tree and obtain a more usable and efficient representation.

In [25], a system is modeled with the help of a particular type of process algebra, called value-passing quality calculus. Given a target location or an asset of interest in the system, an `AND-OR` attack tree representing how the attacker may reach the location or acquire the asset is constructed using SAT solving.

An explicit modeling of the system has also been exploited in [3], where a state-based attack tree is incrementally derived from a quantitative analysis of the transition system representing the real-life system to be analyzed. First, optimal paths, e.g., those corresponding to the cheapest or the fastest attacks, are determined in the transition system, and they are then used to identify leaves that contribute to these optimal attacks and could therefore be interesting candidates for further refinement.

The work described in [9] uses a similar system model as [3]. The states of the transition system are represented by the set of predicates valid in this state. The objective is to generate an attack tree based on a set of successful traces in the transition system, i.e., traces that start from the initial state and end in any state containing a desired set of predicates. The particularity of the obtained tree is that it is refinement-aware, i.e., that its nodes correspond to the meaningful levels of abstraction that can be expressed using the underlying transition system components. It is to be noted that the resulting tree contains only `OR` and `SAND` refinements, i.e., no classical `AND` operator is used. In the worst case, this may imply that the size of the produced tree is exponentially larger than if the `AND` refinement was used.

In all generation approaches described above, attack trees are synthesized for a given system from its formal model. They thus capture only the attacks that apply to this specific system. In contrast, in our framework, an attack tree might be generic, and it is put in context of an, *a priori*, independent system. Our goal is to determine which of the attacks covered by the tree are indeed applicable to the given system, and which of the paths in the system correspond to attacks in practice, but are not covered by the tree. To the best of our knowledge, the problem of missing attacks has not yet been formally investigated.

A second novelty of our work is a formalization of *weak* refinement operators for attack trees. The weak operators capture the fact that some actions, although not explicitly present in a tree, might be required so that the path in the system is indeed an attack. It turns out that a very similar issue has recently been studied by Mantel and Probst in [17], where the authors introduced the *purity* property. This property stipulates whether an attack should perfectly fit a sequence of an attack tree leaves or whether they can be interleaved with other actions. The core problem addressed by our weak refinement operators and by the purity property is the same, but some difference can be observed. Our weak operators are used locally at a given node, so it is possible to accept some additional actions at some but not at all nodes of the tree. In contrast, the purity criterion of [17] seems to be defined as a property of the attack tree semantics, so the additional actions are allowed either at all nodes or at none.

3 Background on attack trees

We start with an explanation of what an attack tree is, in Section 3.1, before introducing the notion of an attack in Section 3.2.

3.1 Attack trees informally

Intuitively speaking, an attack tree is a labeled tree representing how an attacker can proceed to attack a system. The label of the root node describes the main goal of the attacker and the remaining nodes refine this goal into subgoals. In this work, we use classical refinement operators – *disjunctive* (`OR`), *conjunctive* (`AND`), and *sequential* (`SAND`) – as well as new, weak operators – *weak conjunctive* (`wAND`) and *weak sequential* (`wSAND`).

The attack tree leaves represent goals that are precise enough and thus do not need to be refined any further. The goal of an `OR` node is achieved if at least one of the goals of its children is achieved. Achievement of the goal of an `AND` node requires to achieve the goals of all of its children. The goal of a `SAND` node is achieved if the goals of all of its children are achieved in the specified order. To achieve the goal of a node refined using `wAND` (resp. `wSAND`), the attacker needs to achieve the goals of all of its children (resp. in the given order) but in addition, some other actions (not necessarily under the control of the attacker) may also be necessary before the goal of the node can be fully reached. For instance, consider that to be able to attack a system, the attacker needs to deactivate several alarms. In order to do so, he shuts down the power supply, and pursues his attack. However, after a short period of time, the back-up power supply takes over and the alarms are back on. The action of putting the electricity back on is not the attacker's action, so it will not appear in an attack tree explicitly. To make modeling of such attack scenarios possible, we use the weak refinement operators.

Example 1. An example of an attack tree is given in Fig. 1, where we use standard notation: arcs denote conjunctive nodes, and arrows sequential ones. We will later use dotted arcs and arrows for weak operators. The main goal of the attacker is to get a document. To achieve this, the attacker may either corrupt an employee, or steal the document by himself. To corrupt the employee, the attacker may bribe or blackmail him. Stealing the document requires penetrating the building and taking the document. To enter the building, the attacker must unlock the door and then enter undetected. The lock can be opened with a key that would need to be stolen or it can be forced. The attacker enters undetected if he manages to deactivate the alarm and pass the door.

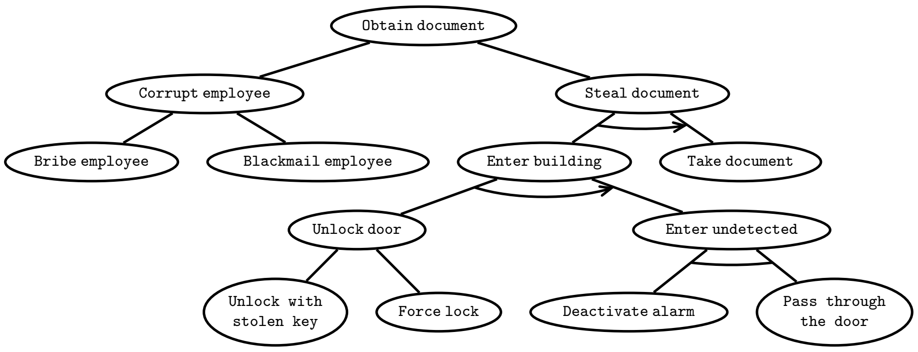


Fig. 1: An attack tree for obtaining a document.

We now give the definition and formal notation for attack trees. For the rest of the paper, we fix a set `ACT` that models all the actions that the attacker can execute.

Definition 1. *An* attack tree θ *over* `ACT` *is either a leaf* $a \in$ `ACT`*, or a composed tree* $\texttt{OP}(\theta_1, \theta_2, \ldots, \theta_n)$*, where* $\texttt{OP} \in \{\texttt{OR}, \texttt{SAND}, \texttt{AND}, \texttt{wSAND}, \texttt{wAND}\}$ *and* $\theta_1, \theta_2, \ldots, \theta_n$ *are attack trees over* `ACT`*.*

To differentiate classical `OR`, `AND`, and `SAND` refinement operators from `wAND` and `wSAND`, we refer to the former ones as *strong* and to the latter ones as *weak*.

3.2 Attacks

An attack tree represents a collection of attacks that an attacker can follow to achieve his goal. In this work we formalize an *attack* as a sequence of the elements of `ACT` sufficient to achieve the goal of the root node of the tree, accordingly to the refinement operators.

Example 2. For instance, the tree in Fig. 1 models the following set of attacks:

A_1 : `Bribe employee`
A_2 : `Blackmail employee`
A_3 : `Unlock with stolen key`, `Deactivate alarm`, `Pass through the door`, `Take document`
A_4 : `Unlock with stolen key`, `Pass through the door`, `Deactivate alarm`, `Take document`
A_5 : `Force lock`, `Deactivate alarm`, `Pass through the door`, `Take document`
A_6 : `Force lock`, `Pass through the door`, `Deactivate alarm`, `Take document`

One can notice that attacks A_3 and A_4 (similarly A_5 and A_6) differ only in the order in which the actions of passing through the door and deactivating the alarm are executed. This is due to the `AND` operator refining the `Enter undetected` node. However, given a specific building where the document is stored, one of these attacks might actually be unfeasible. Indeed, some alarms can be deactivated only from outside, whereas others require a person to first enter the building and then deactivate the alarm within a short, predefined time lapse. Since attack trees are often created without having full knowledge of the system they will be applied to, they might represent some sequences of actions that in reality are not valid attacks. On the contrary, the attack tree modeler who happens to have a specific type of alarm at his home, e.g., the one to be deactivated from outside, might be biased during the attack tree creation process and model `Enter undetected` with `SAND` (`Deactivate alarm`, `Pass through the door`), in which case the sequences A_4 and A_6 will not be considered as possible attacks in the tree, even if they are feasible in the system.

In order to deal with such discrepancies, it is necessary to put an attack tree in the context of the analyzed system to define a formal semantics for attack trees that captures all valid attacks, and only them. We achieve this by an explicit modeling of the analyzed system, using labeled transition systems.

4 Enhancement of the attack tree model

In Section 4.1, we recall basic knowledge on labeled transition systems and define operations on paths that we use in Section 4.2 to equip attack trees with formal semantics relative to the analyzed system.

4.1 System modeling with labeled transition systems

To model real-life systems whose security we want to analyze, we use transition systems with transitions labeled by the elements of `ACT`. We use non-deterministic systems to be able to capture the fact that some actions of the attacker are guided by the environment or are conditioned on the actions of other parties. Exploiting non-determinism to reason about an impact of the environment on an agent behavior is a standard approach in the model checking community (see for example [5, page 22]).

Definition 2. *A* finite transition system *labeled by `ACT` is a tuple $\mathcal{S} \overset{def}{=} (S, \rightarrow)$, where S is the set of states and $\rightarrow \subseteq S \times \mathtt{ACT} \times S$ is the set of transitions.*

Note that we write $s_1 \xrightarrow{a} s_2$ instead of $(s_1, a, s_2) \in \rightarrow$ when referring to a transition between two states $s_1, s_2 \in S$ labeled by $a \in \mathtt{ACT}$, and call this transition an a-transition.

An example of a very simple transition system, modeling how a person can deactivate two alarms, is given in Fig. 5, in Section 4.2.

Let us now recall the notion of a path in a transition system, that we use in our framework to formalize attacks.

Definition 3. *A* path π *in a transition system* $\mathcal{S}$ *is a finite sequence of the form* $\pi = s_0 a_1 s_1 \dots a_n s_n$*, where, for all* $0 \leq i < n$*, we have* $s_i \xrightarrow{a_{i+1}} s_{i+1}$*. We let* $\pi.first \overset{def}{=} s_0$*,* $\pi.last \overset{def}{=} s_n$*, and* $trace(\pi) \overset{def}{=} a_1 \dots a_n \in \mathtt{ACT}^*$*.*

A path $\pi = s_0 a_1 s_1 \dots a_n s_n$ *is* elementary *whenever the states occurring in* π *are all distinct, namely, for all* $i \neq j$*, we have* $s_i \neq s_j$*.*

The set of all paths (resp. elementary paths) in $\mathcal{S}$ is denoted by $\Pi(\mathcal{S})$ (resp. $\Pi_{elem}(\mathcal{S})$). Given a set of paths $\Pi \subseteq \Pi(\mathcal{S})$, we write $\Pi.first$ for the set $\{\pi.first \mid \pi \in \Pi\}$ and $\Pi.last$ for the set $\{\pi.last \mid \pi \in \Pi\}$.

The size of a path π, written $|\pi|$, is equal to its number of transitions, and $\pi(i)$ is the $i+1^{st}$ state of π. We have $\pi(0) = \pi.first$ and $\pi(|\pi|) = \pi.last$. The subsequence of states in a path π from $\pi(i)$ to $\pi(j)$ is denoted by $\pi[i,j]$. Such a subsequence is called a *factor* of π with *anchoring* $[i,j]$.

In order to further analyze paths, we now define their concatenation and parallel composition. Intuitively speaking, the concatenation of two paths π_1 and π_2 can be done if the last state of π_1 is equal to the first state of π_2. The result of the concatenation is then a path π containing the sequence of states of π_1 followed by the sequence of states of π_2, without any additional state or transition, as illustrated in Fig. 2.

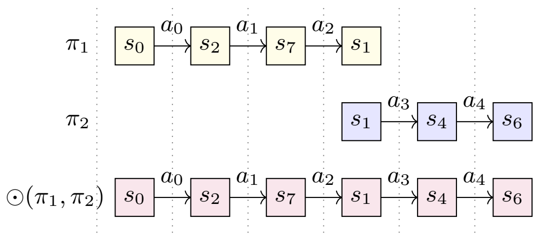


Fig. 2: Concatenation of paths π_1 and π_2.

Formally, the path concatenation is defined as follows:

Definition 4. *Let* $\pi_1, \pi_2, \dots, \pi_n \in \Pi(\mathcal{S})$ *be paths in* $\mathcal{S}$*, such that* $\pi_i.last = \pi_{i+1}.first$*, for* $1 \leq i < n$*. The concatenation of* $\pi_1, \pi_2, \dots, \pi_n$*, denoted with* $\odot(\pi_1, \pi_2, \dots, \pi_n)$*, is the path* π *satisfying* $\pi[\sum_{k=1}^{i-1} |\pi_k|, (\sum_{k=1}^{i-1} |\pi_k|) + |\pi_i|] = \pi_i$*, for every* $i \in \{1, \dots, n\}$*.*

By extension, given sets of paths $\Pi_1, \Pi_2, \dots, \Pi_n$*, we let*

$$\odot(\Pi_1, \Pi_2, \dots, \Pi_n) \overset{def}{=} \{\odot(\pi_1, \pi_2, \dots, \pi_n) \mid \pi_i \in \Pi_i, \ for \ \ 1 \leq i \leq n\}.$$

Concatenation on sets of paths will be used to define the semantics of the `SAND` operator in attack trees. To formalize the `AND` operator, we will use the notion of *parallel composition* of paths.

Intuitively speaking, a path π is a parallel composition of paths $\pi_1, \pi_2, \ldots, \pi_n$ if it is possible to obtain π by combining π_1, π_2, $\ldots$, π_n in a way where every pair of consecutive states (i.e., every transition) in π is covered by one of the π_i. Thus, the very action carried by this transition takes part in at least one of the paths among $\pi_1, \pi_2, \ldots, \pi_n$. Fig. 3 illustrates the notion of parallel composition of paths.

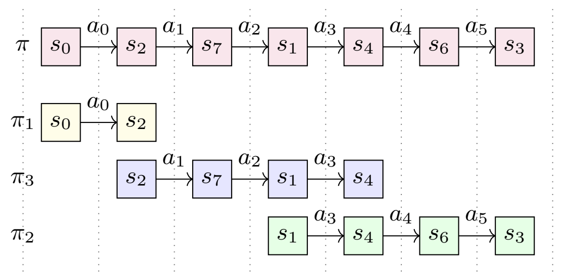

π: $s_0 \xrightarrow{a_0} s_2 \xrightarrow{a_1} s_7 \xrightarrow{a_2} s_1 \xrightarrow{a_3} s_4 \xrightarrow{a_4} s_6 \xrightarrow{a_5} s_3$
π_1: $s_0 \xrightarrow{a_0} s_2$
π_3: $s_2 \xrightarrow{a_1} s_7 \xrightarrow{a_2} s_1 \xrightarrow{a_3} s_4$
π_2: $s_1 \xrightarrow{a_3} s_4 \xrightarrow{a_4} s_6 \xrightarrow{a_5} s_3$

Fig. 3: Path π is a parallel composition of paths π_1, π_2, and π_3.

Fig. 4 shows an example of a path π that is *not* a parallel composition of π_1, π_2, π_3, because the transition $s_2 \xrightarrow{a_1} s_7$ in π is not covered by any of the π_i's.

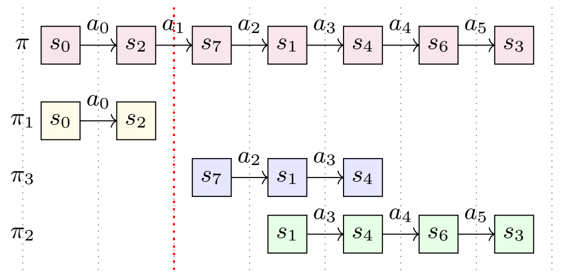

π: $s_0 \xrightarrow{a_0} s_2 \xrightarrow{a_1} s_7 \xrightarrow{a_2} s_1 \xrightarrow{a_3} s_4 \xrightarrow{a_4} s_6 \xrightarrow{a_5} s_3$
π_1: $s_0 \xrightarrow{a_0} s_2$
π_3: $s_7 \xrightarrow{a_2} s_1 \xrightarrow{a_3} s_4$
π_2: $s_1 \xrightarrow{a_3} s_4 \xrightarrow{a_4} s_6 \xrightarrow{a_5} s_3$

Fig. 4: Path π is not a parallel composition of π_1, π_2, π_3.

In Example 3 of Section 4.2, we expose the reasons for introducing such a parallel composition of paths, whose formal definition is as follows.

Definition 5. *A path π is a* parallel composition *of paths $\pi_1, \pi_2, \ldots, \pi_n$, denoted by $\pi \in /\!\!/\!\!\backslash(\pi_1, \pi_2, \ldots, \pi_n)$, whenever the following conditions are satisfied:*

- *for every $i \in \{1, \ldots, n\}$, the path π_i is a factor of π at some anchoring $[k_i, l_i]$,*
- *for every $j \in \{0, \ldots, |\pi| - 1\}$, the inclusion $[j, j+1] \subseteq [k_i, l_i]$ holds, for some $i \in \{1, \ldots, n\}$.*

By extension, given sets of paths $\Pi_1, \Pi_2, \ldots, \Pi_n$, we let

$$/\!\!/\!\!\backslash(\Pi_1, \Pi_2, \ldots, \Pi_n) \overset{def}{=} \{/\!\!/\!\!\backslash(\pi_1, \pi_2, \ldots, \pi_n) \mid \pi_i \in \Pi_i, \text{ for } 1 \leq i \leq n\}.$$

It is important to notice that our parallel composition of paths is orthogonal to the notion of parallel composition of labeled transition systems: parallel composition of paths defined in Definition 5 captures the concomitance of goal achievements along a fixed execution (path) of the transition system, while the parallel composition of transition systems reflects the concomitance of executions.

Finally, to formalize the weak refinement operators, we relax the parallel composition to its weak form, called *weak parallel composition*. In this case, the paths to be composed do not need to overlap at all.

Definition 6. *Path* π *is a* weak parallel composition *of paths* $\pi_1, \pi_2, \ldots, \pi_n$, *denoted by* $\pi \in |||(\pi_1, \pi_2, \ldots, \pi_n)$, *whenever the following holds:*

- *for every* $i \in \{1, \ldots, n\}$, *the path* π_i *is a factor of* π *at some anchoring* $[k_i, l_i]$,
- $\pi[0, |\pi_i|] = \pi_i$, *for some* $i \in \{1, \ldots, n\}$,
- $\pi[|\pi| - |\pi_j|, |\pi|] = \pi_j$, *for some* $j \in \{1, \ldots, n\}$.

By extension, given sets of paths $\Pi_1, \Pi_2, \ldots, \Pi_n$, *we let*

$$|||(\Pi_1, \Pi_2, \ldots, \Pi_n) \overset{def}{=} \{|||(\pi_1, \pi_2, \ldots, \pi_n) \mid \pi_i \in \Pi_i, \ for \ \ 1 \leq i \leq n\}.$$

Note that in Fig. 4, while π is not a parallel composition of paths π_1, π_2, π_3, it is a weak parallel composition, i.e., $\pi \in |||(\pi_1, \pi_2, \pi_3)$.

In the next section, we use the operations on sets of paths defined here to construct the semantics of an attack tree in the presence of a system.

4.2 Attack tree semantics in the presence of a system model

Let $\mathcal{S}$ be a transition system labeled by $\texttt{ACT}$, and let θ be an attack tree over $\texttt{ACT}$. Our objective is to define the semantics of θ in terms of sequences of actions from $\mathcal{S}$, that satisfy the root goal of θ. Each node will thus be interpreted as a set of paths in $\mathcal{S}$, that is constructed as follows.

A leaf node labeled with $a \in \texttt{ACT}$ is simply interpreted with paths of length one, corresponding to a-transitions in $\mathcal{S}$. The interpretation of an `OR` node is the union of the sets of paths corresponding to its children. Indeed, any path satisfying the goal of a child of an `OR` node also satisfies the goal of the `OR` node itself.

To achieve the goal of a `SAND` node, the attacker needs to achieve the goals of all of its children in the given order. Thus, to provide the interpretation of a `SAND` node, we concatenate the sets of paths corresponding to its children, as defined in Definition 4. Similarly, to achieve the goal of a `wSAND` node, the goals of all of its children need to be achieved in the given order, but arbitrary other actions can occur between each subgoal realization. This permits to capture the interleaving of this sequential goal with other parts of the attack, as well as adequately model cases where some system reaction or behavior is needed to continue the attack (recall the example of the back-up power supply, discussed in Section 3.1). Formally, we thus interpret a `wSAND` node using concatenation of sets of paths satisfying the node's children interleaved with any path possible in the system.

Finally, `AND` and `wAND` require that the goals of all of their child nodes are achieved, but do not impose any order on this achievement. In the case of the strong operator `AND`, the actions of each subgoal must be seen contiguously, whereas for the weak operator `wAND` these subgoals can be arbitrarily interleaved with other actions. To capture this behavior, parallel (Definition 5) and weak parallel composition (Definition 6) of the sets of paths are used to provide the interpretation for the two types of conjunctively refined nodes.

Overall weak versions of the operators are more lenient than their strong counterparts. They introduce a weaker notion of precedence that fits the use of underspecified attack trees (some actions of the system are not precisely modeled), and more generally the fact that the immediate "Next" constraint of the strong operators is difficult to enforce in the context of a concurrent system. These weak operators let us express "stutter invariant" behavior with attack trees. In practice, for concurrent systems, stutter invariant property specification is often more relevant than using the full temporal logic with neXt. Definition 7 summarizes the above discussion.

Definition 7. *Given a transition system $\mathcal{S}$, the* path semantics *of an attack tree θ is the set of paths $\llbracket\theta\rrbracket^{\mathcal{S}} \subseteq \Pi(\mathcal{S})$, defined by induction as follows:*

- $\llbracket a\rrbracket^{\mathcal{S}} = \{s_1 a s_2 \in \Pi(\mathcal{S}) \mid s_1, s_2 \in S\}$,
- $\llbracket \mathtt{OR}(\theta_1, \theta_2, \ldots, \theta_n)\rrbracket^{\mathcal{S}} = \cup(\llbracket\theta_1\rrbracket^{\mathcal{S}}, \llbracket\theta_2\rrbracket^{\mathcal{S}}, \ldots, \llbracket\theta_n\rrbracket^{\mathcal{S}})$,
- $\llbracket \mathtt{SAND}(\theta_1, \theta_2, \ldots, \theta_n)\rrbracket^{\mathcal{S}} = \odot(\llbracket\theta_1\rrbracket^{\mathcal{S}}, \llbracket\theta_2\rrbracket^{\mathcal{S}}, \ldots, \llbracket\theta_n\rrbracket^{\mathcal{S}})$,
- $\llbracket \mathtt{AND}(\theta_1, \theta_2, \ldots, \theta_n)\rrbracket^{\mathcal{S}} = \wedge\!\!\!\wedge\!\!\!\wedge(\llbracket\theta_1\rrbracket^{\mathcal{S}}, \llbracket\theta_2\rrbracket^{\mathcal{S}}, \ldots, \llbracket\theta_n\rrbracket^{\mathcal{S}})$,
- $\llbracket \mathtt{wSAND}(\theta_1, \theta_2, \ldots, \theta_n)\rrbracket^{\mathcal{S}} = \odot(\llbracket\theta_1\rrbracket^{\mathcal{S}}, \Pi(\mathcal{S}), \llbracket\theta_2\rrbracket^{\mathcal{S}}, \Pi(\mathcal{S}), \ldots, \Pi(\mathcal{S}), \llbracket\theta_n\rrbracket^{\mathcal{S}})$,
- $\llbracket \mathtt{wAND}(\theta_1, \theta_2, \ldots, \theta_n)\rrbracket^{\mathcal{S}} = |||(\llbracket\theta_1\rrbracket^{\mathcal{S}}, \llbracket\theta_2\rrbracket^{\mathcal{S}}, \ldots, \llbracket\theta_n\rrbracket^{\mathcal{S}})$.

The example below illustrates the use of parallel composition of paths to interpret an `AND` node.

Example 3. Suppose that an attacker needs to deactivate two alarms. To do so, he can either disable Alarm 1 and Alarm 2 in any order, or simply shut down the power supply, which automatically deactivates both alarms. The transition system modeling these possibilities is given in Fig. 5. Suppose that the

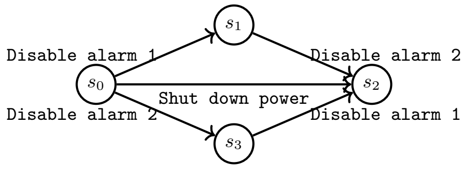


Fig. 5: Simple transition system.

attacker's behavior is modeled with the attack tree from Fig. 6. The path semantics of this tree is composed of paths $s_0 s_1 s_2$, $s_0 s_3 s_2$, and $s_0 s_2$ (we omit the actions for readability). The last path is interesting. Indeed, $s_0 s_2$ belongs to both $[\![\texttt{Deactivate Alarm 1}]\!]^{\mathcal{S}}$ and $[\![\texttt{Deactivate Alarm 2}]\!]^{\mathcal{S}}$. It is also a valid path in the parallel composition $/\!\!/\!\!\backslash([\![\texttt{Deactivate Alarm 1}]\!]^{\mathcal{S}}, [\![\texttt{Deactivate Alarm 2}]\!]^{\mathcal{S}})$, because it satisfies both subgoals at the same time. This example shows that, in the parallel composition of paths, and thus in the semantics of `AND` nodes, any overlap between the paths interpreting their child nodes is allowed, as already illustrated in Fig. 3.

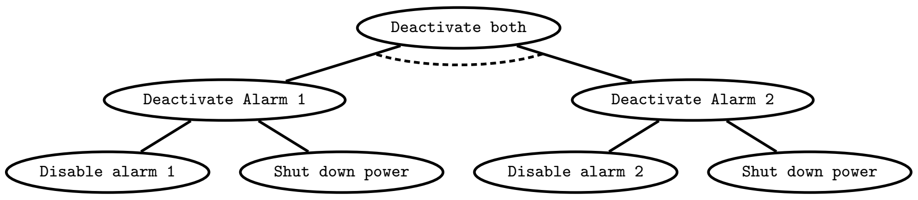


Fig. 6: Example with overlap on `AND` node.

Notice that the framework developed in this work allows for repetitions of an action in the tree, as it is the case of `Shut down power` in Example 3. The presence of such repeated actions may, for instance, result from the use of attack tree libraries. For some scenarios, having an action several times in a tree might be necessary. Recall the power example discussed in Section 3.1, where the back-up power supply is activated short after the main electricity source is switched off. Depending on how quickly the attacker performs his attack, the `Shut down power` action from the tree in Fig. 6 will need to be done either one or two times. To cover both cases, the action is repeated in the tree, and our formalism is flexible enough to interpret such repeated nodes as the same or separated instances of the action.

4.3 System-based approach to classical view on attack trees

To finish this section, we relate the system-based view on attack trees with classical approaches where the system is not considered. Indeed, most of existing semantics for attack trees do not take the analyzed system into account. We would like to point out that our system-based framework can simulate such approaches by considering the *universal transition system over* $\mathtt{ACT}$, written $\mathcal{U}_{\mathtt{ACT}}$, allowing to execute any possible sequence of actions over $\mathtt{ACT}$. The universal transition system is composed of a single state and a looping transition for each action in $\mathtt{ACT}$, hence it looks like a flower. An example of $\mathcal{U}_{\mathtt{ACT}}$ over $\mathtt{ACT} = \{1, \ldots, 9\}$ is given in Fig. 7.

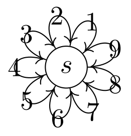


Fig. 7: System $\mathcal{U}_{\{1,\ldots,9\}}$.

To define the semantics of attack trees that is independent of the system, we introduce *trace language* of an attack tree θ over $\texttt{ACT}$, denoted by $\mathcal{L}(\theta) \subseteq \texttt{ACT}^*$, and defined as:

$$\mathcal{L}(\theta) \overset{def}{=} \{trace(\rho) \mid \rho \in [\![\theta]\!]^{\mathcal{U}_{\texttt{ACT}}}\}.$$

The following result makes a link between the trace language of an attack tree and its path semantics.

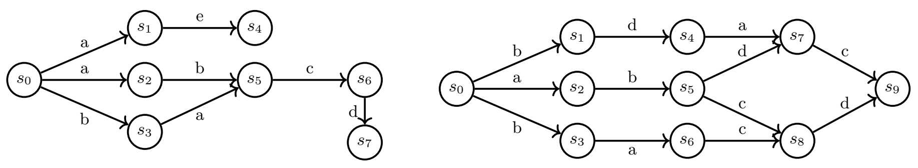
Proposition 1. *Given a transition system $\mathcal{S}$ and a path $\pi \in \Pi(\mathcal{S})$, we have that $\pi \in [\![\theta]\!]^{\mathcal{S}}$ if, and only if, $trace(\pi) \in \mathcal{L}(\theta)$.*

Proof. By definition of $\mathcal{L}(\theta)$, we have to show that $\pi \in [\![\theta]\!]^{\mathcal{S}}$ if, and only if, there exists $\rho \in [\![\theta]\!]^{\mathcal{U}_{\texttt{ACT}}}$ with $trace(\pi) = trace(\rho)$. The candidate for ρ is the path like π but where each state is replaced by the unique state of $\mathcal{U}_{\texttt{ACT}}$; note that ρ is indeed a path of $\mathcal{U}_{\texttt{ACT}}$ and that by construction $trace(\pi) = trace(\rho)$. It is routine to verify by induction over θ that this candidate is adequate. □

The following example emphasizes the role played by the system in the path semantics.

Example 4. Take the tree $\theta = \texttt{SAND}(\texttt{AND}(a, b), \texttt{AND}(c, d))$. It is easy to see that $\mathcal{L}(\theta) = \{abcd, abdc, bacd, badc\}$. However, when interpreting the tree w.r.t. a system, using the semantics from Definition 7, not all sequences of actions will necessarily be feasible.

– If we consider the universal system $\mathcal{U}_{\texttt{ACT}}$, then the path semantics will contain the four attacks.
– For the system $\mathcal{S}_1$ in Fig. 8a, the path semantics of the tree only contains attacks *abcd* and *bacd*.
– Finally, for the system $\mathcal{S}_2$ in Fig. 8b, the path semantics contains attacks *abcd*, *abdc*, and *bacd*.



(a) System $\mathcal{S}_1$ (b) System $\mathcal{S}_2$

Fig. 8: Different systems induce different path semantics.

Example 4 illustrates that the same attack tree does not have to have the same meaning depending on the underlying system. The path semantics allows the experts to resort to some libraries of already designed trees without the inconvenience of checking what is indeed achievable or not in the system they have in mind. A practical example of such reusable attack tree libraries can be found in Chapter 4 of [19].

5 Missing attacks

In this section, we exploit the path semantics to compare the tree with what the attacker can achieve in the system. In Section 5.1, we formally define missing attacks, and in Sections 5.2 to 5.4 we address the decision problem MAE for the existence of missing attacks together with the related trace membership decision problem TATM, and provide complexity bounds for these problems. Finally, Section 5.5 describes a prototype implementation that we developed to automate the solving of the TATM problem.

5.1 The definition of missing attacks

We now explain how we can warn the security experts about possibly missing attacks in an attack tree. We start by an illustrating example. Consider the tree θ from Example 4 and let us look again at system $\mathcal{S}_2$ in Fig. 8b. What can we say about the sequence of actions *bdac*? According to system $\mathcal{S}_2$, this sequence starts and ends in the same states as actual attacks. Thus, the four paths between s_0 and s_9 are somehow equivalent in the system, but one of them is not in the semantics of attack tree θ. It could be interesting for the expert to get a warning about such paths that are equivalent to attacks in the system, but that he did not include in the tree. This is what we call *completeness analysis*.

Note that we are not claiming that any such path in the system *must* be considered as an attack, that is up to the experts to decide. However, we believe that analyzing the completeness of the attack tree and giving warnings to security experts about these paths can prevent some human-related errors, which can be troublesome if not tackled.

To formally define the notion of missing attack, we first introduce the closure of a set of paths that encompasses extra paths having the same extremities as the paths in the set.

Definition 8. *Given a system $\mathcal{S}$, the* closure *of a set of paths $\Pi \in \Pi(\mathcal{S})$ is $cl(\Pi) \overset{def}{=} \{\pi \in \Pi_{elem}(\mathcal{S}) \mid \pi.first \in \Pi.first \text{ and } \pi.last \in \Pi.last\}$.*

It is clear that any elementary path in Π is also in $cl(\Pi)$. However, the reciprocal may not hold in general and this is how we capture missing attacks.

Definition 9. *A path π is a* missing attack *if it belongs to $\Delta_\theta^\mathcal{S} \overset{def}{=} cl(\llbracket\theta\rrbracket^\mathcal{S})\setminus\llbracket\theta\rrbracket^\mathcal{S}$.*

To make it simple, a missing attack in θ is a path in $\mathcal{S}$ that imitates attacks in $\llbracket\theta\rrbracket^\mathcal{S}$ (starts and ends in states of existing attacks) but is not in $\llbracket\theta\rrbracket^\mathcal{S}$ itself. It is important to notice that missing attacks are restricted to elementary paths. This is a robust choice: indeed, if we allowed non-elementary paths, we would basically take an inventory of existing attacks extended with extra uninteresting cycles, irrelevant from the point of view of a rational attacker for adding useless sequences of actions.

Example 5. We illustrate the notion of missing attacks on the transition system from Fig. 5. Suppose that the tree designer is not interested in Alarm 2. We consider the leaf attack tree `Disable alarm 1`. We omit the actions for readability, and we have: $[\![\texttt{Disable alarm 1}]\!]^{\mathcal{S}} = \{s_0 \to s_1, s_3 \to s_2\}$, and $cl([\![\texttt{Disable alarm 1}]\!]^{\mathcal{S}}) = \{s_0 \to s_1,\ s_3 \to s_2,\ s_0 \to s_1 \to s_2,\ s_0 \to s_3 \to s_2,\ s_0 \to s_2\}$, so that $\Delta_{\theta}^{\mathcal{S}} = \{s_0 \to s_1 \to s_2,\ s_0 \to s_3 \to s_2,\ s_0 \to s_2\} \neq \emptyset$. In particular, the missing attack $s_0 \to s_2$ might be problematic if the expert protects the system against the action `Disable alarm 1` only, because there will still be a possibility for the attacker to counter the alarm by shutting down the power supply.

To allow for reasoning about missing attacks, we introduce and investigate the Missing Attack Existence decision problem.

5.2 The Missing Attack Existence problem

Since in risk analysis missing attacks can have severe consequences, we address the natural question of the existence of missing attacks, captured by the following decision problem.

Definition 10 (Missing attack existence problem (MAE)).

– *Input: an attack tree θ and a system $\mathcal{S}$.*
– *Output: $\Delta_{\theta}^{\mathcal{S}} \neq \emptyset$?*

Otherwise said, can we find three paths π, π_1, π_2 that satisfy the following constraints: $\pi.first = \pi_1.first$, $\pi.last = \pi_2.last$, $\pi \notin [\![\theta]\!]^{\mathcal{S}}$, $\pi_1 \in [\![\theta]\!]^{\mathcal{S}}$, and $\pi_2 \in [\![\theta]\!]^{\mathcal{S}}$?

It is very tempting to design a non-deterministic algorithm that can select three paths and then checks these five constraints above. However, due to potential weak operators `wSAND` and `wAND`, it seems difficult to bound the size of these paths. While the size of π can be bounded by the size of the system, as missing attacks are elementary paths, it is unclear how to bound the size of paths π_1 and π_2. Discarding weak operators gives a natural bound which is the number of leaves of θ, thus polynomial in the size of the input.

Now, once these three paths are guessed, the non-deterministic algorithm verifies the five constraints. The first two can be verified in $O(1)$, while the last three reduce to answering the *Trace Attack Tree Membership problem (TATM)* formalized in the following definition.

Definition 11 (Trace Attack Tree Membership problem (TATM)).

– *Input: an attack tree θ (over `ACT`) and a trace $t \in \texttt{ACT}^*$*
– *Output: $t \in \mathcal{L}(\theta)$?*

Proposition 2. *TATM is* NP*-complete.*

Based on Proposition 2 (a corollary of Propositions 3 and 5 fully proven in the next Sections 5.3 and 5.4 respectively), we design the non-deterministic Algorithm 1 that makes three independent calls to NP oracles.

```
Input: An attack tree $\theta$, a system $\mathcal{S}$.
Output: ACCEPT if $\Delta_\theta^{\mathcal{S}} \neq \emptyset$.
CHOOSE elementary path $\pi \in \Pi(\mathcal{S})$, and $\pi_1, \pi_2 \in \Pi(\mathcal{S})$ of length at
  most $|\theta|$;
if $\pi.first \neq \pi_1.first$ or $\pi.last \neq \pi_2.last$ then
    REJECT
end
else
    if the NP oracle for the question "$trace(\pi) \in \mathcal{L}(\theta)$" answers "Yes"
      then
        REJECT
    end
    else
        if the NP oracle for the question "$trace(\pi_1) \in \mathcal{L}(\theta)$" answers
          "No" then
            REJECT
        end
        else
            if the NP oracle for the question "$trace(\pi_2) \in \mathcal{L}(\theta)$" answers
              "No" then
                REJECT
            end
            else
                ACCEPT
            end
        end
    end
end
```

Algorithm 1: $MissingAttack(\theta, \mathcal{S})$.

Correctness of Algorithm 1, i.e., the fact that it can return **ACCEPT** if, and only if, $\Delta_\theta^{\mathcal{S}} \neq \emptyset$, is easy to establish. Indeed, assume Algorithm 1 can return **ACCEPT**. Then, by Proposition 1, there is a way to choose $\pi \notin [\![\theta]\!]^{\mathcal{S}}$, $\pi_1 \in [\![\theta]\!]^{\mathcal{S}}$ and $\pi_2 \in [\![\theta]\!]^{\mathcal{S}}$, such that $\pi.first = \pi_1.first$ and $\pi.last = \pi_2.last$. This shows that π is a missing attack, so that $\Delta_\theta^{\mathcal{S}} \neq \emptyset$. Reciprocally, if $\Delta_\theta^{\mathcal{S}} \neq \emptyset$, pick a missing attack $\pi \in \Delta_\theta^{\mathcal{S}}$. By definition, $\pi \notin [\![\theta]\!]^{\mathcal{S}}$ and there must exist $\pi_1, \pi_2 \in [\![\theta]\!]^{\mathcal{S}}$ with $\pi.first = \pi_1.first$ and $\pi.last = \pi_2.last$, so Algorithm 1 can return **ACCEPT** by non-deterministically choosing these three paths, which concludes.

Corollary 1. *The problem* MAE *restricted to attack trees with operators ranging over* $\{\texttt{OR}, \texttt{SAND}, \texttt{AND}\}$ *is in* Σ_2^P *of the polynomial hierarchy.*

We recall that Σ_2^P is the class of problems that can be solved by a non-deterministic polynomial-time algorithm with queries to NP oracles [24].

We are able to establish that MAE is not easy[4] by showing its co-NP-hardness.

Theorem 1. *The problem MAE is* co-NP*-hard, even if we restrict to trees with operators ranging over* $\{\texttt{OR}, \texttt{SAND}, \texttt{AND}\}$ *only.*

Proof. We describe a reduction from TATM to MAE such that an instance of TATM is negative if, and only if, its reduction is a positive instance of MAE. Since Proposition 3 entails the NP-hardness of TATM, we easily conclude.

Let θ (over $\texttt{ACT}$) and $t = a_1a_2 \ldots a_n \in \texttt{ACT}^*$ be an instance of TATM. Pick a fresh action symbol $\# \notin \texttt{ACT}$. We define the attack tree $\theta' \overset{def}{=} \texttt{OR}(\theta, \#)$ and the transition system $\mathcal{S}$ with $n+1$ states $s_0, s_1, \ldots s_n$ with only two paths: the path $\pi_t \overset{def}{=} s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ and the path $s_0 \xrightarrow{\#} s_n$. It is easy to prove that $\pi_t \in \Delta_{\theta'}^{\mathcal{S}}$ if, and only if, $t \notin \mathcal{L}(\theta)$, which concludes. □

We do not know if MAE is Σ_2^P-hard, we unsuccessfully attempted to reduce the typical Σ_2^P-complete problem QBF_2 that asks if a quantified Boolean formula of the form $\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m \varphi$, where φ is a formula over variables $x_1, \ldots, y_m$, evaluates to true.

We now come back to proving Proposition 2 regarding the NP-completeness of TATM, and show it in the two next sections.

5.3 The NP-hardness of TATM

This section is dedicated to show that the TATM problem is NP-hard.

Proposition 3. *TATM is* NP*-hard, even if we discard weak operators* $\texttt{wSAND}$ *and* $\texttt{wAND}$*.*

We consider the decision problem of *Packed Interval Covering* (PIC), which is NP-complete according to [22].

Let N be an integer. The PIC problem consists in deciding whether we can cover (in the classical sense) interval $[1, N]$ by selecting exactly one subinterval per pack of subintervals given as input. For instance, if the packs are $P_1 = \{[1,6],[5,9]\}$, $P_2 = \{[1,3],[4,6],[7,7]\}$, $P_3 = \{[4,4]\}$, we can cover $[1,9]$ by selecting $[5,9]$, $[1,3]$ and $[4,4]$, as illustrated in Fig. 9.

[4] This holds under the assumption that P $\neq$ NP.

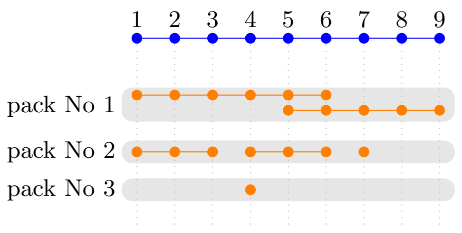


Fig. 9: An instance of the PIC problem.

Formally, we state the following definition.

Definition 12 (Packed Interval Covering (PIC) problem).

- *Input: an integer $N > 0$ and a family of finite sets $P_1, \ldots, P_M$ (*packs*) of subintervals of* $[1, N]$.
- *Output: are there subintervals $I_1 \in P_1, \ldots, I_M \in P_M$, such that $\bigcup_{k=1}^{M} I_k =$* $[1, N]$*?*

Proposition 4. *PIC is* NP*-complete.*

Arguing that PIC belongs to NP is easy. The certificate is simply a list of M sub-intervals of $[1, N]$ (given by their bounds), thus it is polynomial in the size of the packs input. It remains to verify for $i \in \{1, \ldots, M\}$ that interval of rank i belongs to pack P_i, and that the union of the chosen intervals covers $[1, N]$, which can all be performed in polynomial time. Regarding the NP-hardness of PIC, there is a polynomial reduction from the NP-complete problem (3, B2)-SAT [6], which is the restriction of 3-SAT where each variable has exactly two positive occurrences and two negative occurrences. The full proof of this reduction can be found in [22].

Back to the proof of Proposition 3, we exhibit a polynomial reduction of PIC into TATM, so TATM is NP-hard. This reduction relies on a transformation that, given an instance I of PIC, returns an instance I' of TATM, of size polynomial in the size of I, and such that I is a positive instance of PIC iff I' is a positive instance of TATM.

Instead of providing the full proof, we illustrate the idea on the example of the 3 packs $P_1 = \{[1, 6], [5, 9]\}$, $P_2 = \{[1, 3], [4, 6], [7, 7]\}$, $P_3 = \{[4, 4]\}$ to cover the whole interval $[1, 9]$. Its corresponding instance of TATM is composed of the attack tree θ_0, depicted in Fig. 10 (we allow unary OR and SAND operators, as for nodes labeled P_3 and $[4, 4]$ on the right-hand side of the tree, for instance), and the trace $t^0 = 123456789$.

Recall that this pack instance has solution $[5, 9], [1, 3], [4, 4]$.

Consider the universal transition system $\mathcal{U}_{\{1,\ldots,9\}}$, and the unique path π_0 whose trace is t^0, namely the path $s \xrightarrow{1} s \xrightarrow{2} s \ldots \xrightarrow{9} s$.

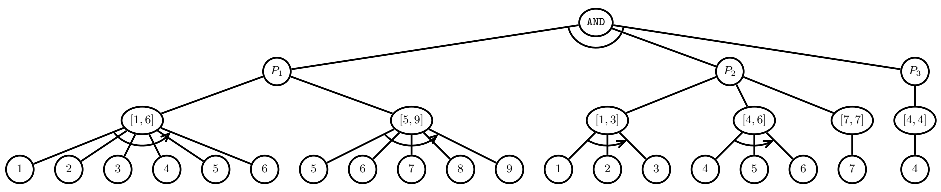


Fig. 10: The tree θ_0.

According to the path semantics,

- π_0 is a parallel composition of paths $\pi_1 = s \xrightarrow{5} s \xrightarrow{6} s \xrightarrow{7} s \xrightarrow{8} s \xrightarrow{9} s$, $\pi_2 = s \xrightarrow{1} s \xrightarrow{2} s \xrightarrow{3} s$, and $\pi_3 = s \xrightarrow{4} s$, and
- π_1 (resp. π_2, π_3) belong to the path semantics of the left (resp. middle, right) subtree of the root node of θ_0.

As a consequence, $t^0 \in \mathcal{L}(\theta_0)$.

Notice that the proposed reduction yields attack trees that do not use weak operators, so that the NP-hardness of TATM holds even if we discard the use of weak opertors in the input tree of the problem.

In order to achieve the proof of Proposition 2, namely to show that TATM is in NP, we have designed a non-deterministic polynomial-time algorithm that is explained in the next section.

5.4 The NP-membership of TATM

Proposition 5. *TATM is in* NP*.*

The proof of Proposition 5 is a direct consequence of the non-deterministic algorithm Algorithm 2 that reads the input trace while marking the nodes of the tree that have been "satisfied" by the prefix trace read so far. Algorithm 2 relies on the subroutine Algorithm 3, with initial call $check(\theta, t, \emptyset, startNodes(\theta), \emptyset)$.

```
Input: An attack tree $\theta$ (over ACT) and a non-empty trace $t \in \mathtt{ACT}^*$
Output: ACCEPT if $t \in \mathcal{L}(\theta)$, REJECT otherwise.
$check(\theta, t, \emptyset, startNodes(\theta), \emptyset)$
```

Algorithm 2: $checkMembership(\theta, t)$.

Before explaining the subroutine $check(\theta, t, \mathtt{Must}, \mathtt{May}, \mathtt{Marked})$ (Algorithm 3), let us first fix the notation used.

We use $Nodes(\theta)$ to denote the set of nodes of the tree θ, and we write $root(\theta)$ for its root, and $Leaves(\theta)$ for its leaf node set. For a node $\gamma \in Nodes(\theta)$, we use self-explanatory notation: $children(\gamma)$, $parent(\gamma)$, and $ancestors(\gamma)$ (including

γ itself). We also consider $sib(\gamma)$ that denote the siblings of γ (excluding γ itself) and $rsib(\gamma)$ that refers to the right sibling of γ (if any). For $\gamma \in Leaves(\theta)$, we write $actionAt(\gamma)$ for the action that labels this node. Also, we may lift relevant notions to a set of nodes Γ, such as $actionAt(\Gamma)$, $ancestors(\Gamma)$. We let $descendentleaves(\gamma)$ denote the set of leaf nodes of the subtree at node γ, and $startNodes(\theta)$ be the subset of nodes of θ whose labels are actions that the attacks may start with. For example, regarding the attack tree of Fig. 1, the set $startNodes(\theta)$ is composed of the leaf nodes that carry label either `Bribe employee`, or `Blackmail employee`, or `Unlock with stolen key`, or `Force lock`.

More formally, $startNodes(\theta)$ is defined by induction over θ and can be easily computed by a terminal recursive algorithm in linear time: $startNodes(a) = \{a\}$; for every $\texttt{OP} \in \{\texttt{OR}, \texttt{AND}, \texttt{wAND}\}$, $startNodes(\texttt{OP}(\theta_1, \ldots, \theta_n)) = \bigcup(startNodes(\theta_i)$; for every $\texttt{OP} \in \{\texttt{SAND}, \texttt{wSAND}\}$, $startNodes(\texttt{OP}(\theta_1, \ldots, \theta_n)) = startNodes(\theta_1)$.

```
Input: A root node $\theta$, a trace $t$, a set of nodes $\texttt{Must} \subseteq Nodes(\theta)$, a set of leaves
       $\texttt{May} \subseteq Leaves(\theta)$, a set of nodes $\texttt{Marked} \subseteq Nodes(\theta)$
Output: ACCEPT if $t \in \mathcal{L}(\theta)$, REJECT otherwise
if $root(\theta) \in \texttt{Marked}$ and $t = \epsilon$ then
    ACCEPT
end
else
    if $May = \emptyset$ or $t = \epsilon$ then
        REJECT
    end
    else
        CHOOSE $\emptyset \subsetneq \Gamma \subseteq \texttt{May}$ ;
        if $actionAt(\Gamma) \not\subseteq \{t(1), \star\}$ then
            REJECT
        end
        else
            if $Must \not\subseteq ancestors(\Gamma)$ then
                REJECT
            end
            else
                $\texttt{Must} \leftarrow \emptyset$; $\texttt{May} \leftarrow \texttt{May} \setminus \Gamma$; $\texttt{Marked} \leftarrow \texttt{Marked} \cup \Gamma$ ;
                forall $\gamma \in \Gamma$ with $actionAt(\gamma) \neq \star$ do
                    $propagate(\theta, \gamma, \texttt{Must}, \texttt{May}, \texttt{Marked})$
                end
            end
            return $check(\theta, t^{\geq 1}, \texttt{Must}, \texttt{May}, \texttt{Marked})$
        end
    end
end
```

Algorithm 3: $check(\theta, t, \texttt{Must}, \texttt{May}, \texttt{Marked})$.

We now explain Algorithm 3 that takes the following inputs:

– an attack tree θ (over $\texttt{ACT}$), according to Definition 1,
– a trace $t \in \texttt{ACT}^*$,
– a set of nodes $\texttt{Must}$ that has to "progress" at next step, initialized to $\emptyset$ for the first call,
– a set of leaves $\texttt{May}$ that may progress at next step, initialized as $startNodes(\theta)$,
– a set of nodes $\texttt{Marked}$ that have already been "consumed" while reading trace t, initialized to $\emptyset$ for the first call.

The principle of Algorithm 3 is as follows:

1. Choose a set of leaves Γ inside $\texttt{May}$ that contains $\texttt{Must}$, and the label of the chosen leaves matches the first action of the trace t.
2. Put those leaves γ in $\texttt{Marked}$.
3. Update $\texttt{Must}$, $\texttt{May}$ and $\texttt{Marked}$ accordingly by calling $propagate(\theta, \gamma, \texttt{Must}, \texttt{May}, \texttt{Marked})$ (Algorithm 4), for each γ in Γ.
4. Goto 1 with the updated sets $\texttt{Must}$, $\texttt{May}$, and $\texttt{Marked}$ and the next action of the trace.

The call to $propagate(\theta, \gamma, \texttt{Must}, \texttt{May}, \texttt{Marked})$ allows us to update the sets $\texttt{May}$, $\texttt{Must}$ and $\texttt{Marked}$, with the consequences of marking the chosen leaves. We set internal nodes as marked according to the path semantics: when a child of an $\texttt{OR}$ node is marked so is this node, and when all the children of either of $\texttt{SAND}$, $\texttt{wSAND}$, $\texttt{AND}$, or a $\texttt{wAND}$ node are marked, so is this node. Also, we put a leaf in $\texttt{May}$ when it gets enabled: namely, when the first child of a $\texttt{wSAND}$ or $\texttt{SAND}$ node is marked, its right sibling (if any) gets enabled. Finally, we add nodes to $\texttt{Must}$ when they are expected to progress at the next step: for SAND nodes once a child is marked, the next child has to progress in the next step, and for an $\texttt{AND}$ node one of its non-marked children has to progress in the next step. The propagation of all these constraints is recursive: at each newly marked node, *propagate* is called again on this node.

The tricky part in this algorithm is due to weak operators: we have to take into account that some actions in t may not be due to any leaf of the tree. This is done by dynamically adding and removing artificial leaves to $\texttt{wSAND}$ and $\texttt{wAND}$ nodes, with special label $\star$, so that these new leaves can be chosen to validate the $\texttt{Must}$ requirements. These added leaves somehow correspond to an "ignore" instruction while reading the trace: if there are some weak operators in the tree and we encounter an action in t that is not a leaf of θ, the algorithm can simply choose to ignore it and keep on with the next action of t, while validating the $\texttt{Must}$ requirement on the parent node.

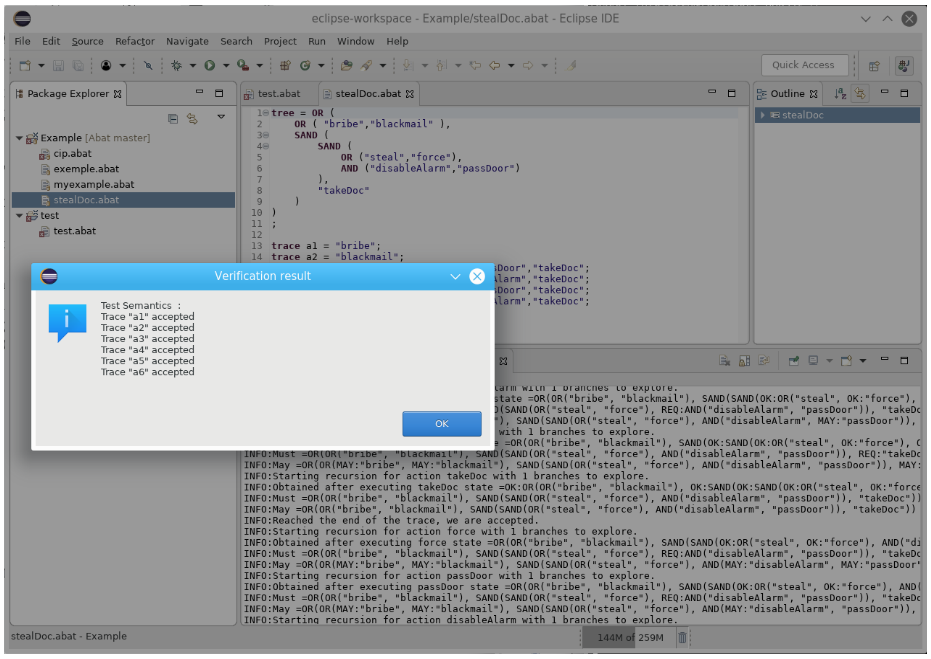
Finally, trace t is in θ if we manage to read the whole trace, and if the root of the tree is marked in the end. Otherwise, the trace is rejected.

```
Input: A root node $\theta$, Must $\subseteq Nodes(\theta)$, May $\subseteq Leaves(\theta)$, Marked $\subseteq Nodes(\theta)$,
       and node $\gamma$ newly added to Marked
Output: Update of Must, May, and Marked
if $\gamma \neq root(\theta)$ then
    $\mu \leftarrow parent(\gamma)$;
    switch $\mu$.OP do
        case OR do
            May $\leftarrow$ May $\setminus descendentleaves(\mu)$; $propagate(\theta, \texttt{Must}, \texttt{May}, \texttt{Marked}, \mu)$
        end
        case SAND do
            if $rsib(\gamma)$ exists then
                $\gamma' \leftarrow rsib(\gamma)$; Must $\leftarrow$ Must $\cup \{\gamma'\}$; May $\leftarrow$ May $\cup\, startNodes(\gamma')$
            end
            else
                $propagate(\theta, \texttt{Must}, \texttt{May}, \texttt{Marked}, \mu)$
            end
        end
        case AND do
            if $sib(\gamma) \subseteq$ Marked then
                $propagate(\theta, \texttt{Must}, \texttt{May}, \texttt{Marked}, \mu)$
            end
            else
                Must $\leftarrow$ Must $\cup \{\mu\}$
            end
        end
        case wSAND do
            if $rsib(\gamma)$ exists then
                $\mu \leftarrow \mu.addchildwithlabel(\star)$;
                May $\leftarrow$ May $\cup \{\mu\}$; May $\leftarrow$ May $\cup \{startNodes(rsib(\gamma))\}$
            end
            else
                forall $\gamma' \in children(\mu)$ with $actionAt(\gamma') = \star$ do
                    May $\leftarrow$ May $\setminus \{\gamma'\}$; $\mu.removechild(\gamma')$
                end
                Marked $\leftarrow$ Marked $\cup \{\mu\}$; $propagate(\theta, \texttt{Must}, \texttt{May}, \texttt{Marked}, \mu)$
            end
            case wAND do
                if $sib(\gamma) \subseteq$ Marked then
                    forall $\gamma' \in children(\mu)$ with $actionAt(\gamma') = \star$ do
                        May $\leftarrow$ May $\setminus \{\gamma'\}$; $\mu.removechild(\gamma')$
                    end
                    Marked $\leftarrow$ Marked $\cup \{\mu\}$; $propagate(\theta, \texttt{Must}, \texttt{May}, \texttt{Marked}, \mu)$
                end
                else
                    $\mu \leftarrow \mu.addchildwithlabel(\star)$; May $\leftarrow$ May $\cup \{\mu\}$
                end
            end
        end
    end
end
```

Algorithm 4: $propagate(\theta, \texttt{Must}, \texttt{May}, \texttt{Marked}, \gamma)$.

5.5 Implementation

To support our approach, we have implemented a proof of concept implementation to solve the TATM problem. The tool is available online as an open source prototype, at `https://github.com/yanntm/Abat`. The implementation includes a small Xtext based editor that allows to specify an attack tree and a set of traces. The tool then checks the membership of each trace in the language of the specified tree, as visualized in Fig. 11.



```
tree = OR (
    OR ( "bribe","blackmail" ),
    SAND (
        SAND (
            OR ("steal","force"),
            AND ("disableAlarm","passDoor")
        ),
        "takeDoc"
    )
)
;

trace a1 = "bribe";
trace a2 = "blackmail";
sDoor","takeDoc";
Alarm","takeDoc";
sDoor","takeDoc";
Alarm","takeDoc";
```

```
Test Semantics :
Trace "a1" accepted
Trace "a2" accepted
Trace "a3" accepted
Trace "a4" accepted
Trace "a5" accepted
Trace "a6" accepted
```

```
state =OR(OR("bribe", "blackmail"), SAND(SAND(OK:OR("steal", OK:"force"),
D(SAND(OR("steal", "force"), REQ:AND("disableAlarm", "passDoor")), "takeDc
"), SAND(SAND(OR("steal", "force"), AND("disableAlarm", MAY:"passDoor")),
 with 1 branches to explore.
e =OR(OR("bribe", "blackmail"), SAND(OK:SAND(OK:OR("steal", OK:"force"), C
INFO:Must =OR(OR("bribe", "blackmail"), SAND(SAND(OR("steal", "force"), AND("disableAlarm", "passDoor")), REQ:"takeDc
INFO:May =OR(OR(MAY:"bribe", MAY:"blackmail"), SAND(SAND(OR("steal", "force"), AND("disableAlarm", "passDoor")), MAY:
INFO:Starting recursion for action takeDoc with 1 branches to explore.
INFO:Obtained after executing takeDoc state =OK:OR(OR("bribe", "blackmail"), OK:SAND(OK:SAND(OK:OR("steal", OK:"force
INFO:Must =OR(OR("bribe", "blackmail"), SAND(SAND(OR("steal", "force"), AND("disableAlarm", "passDoor")), "takeDoc"))
INFO:May =OR(OR("bribe", "blackmail"), SAND(SAND(OR("steal", "force"), AND("disableAlarm", "passDoor")), "takeDoc"))
INFO:Reached the end of the trace, we are accepted.
INFO:Starting recursion for action force with 1 branches to explore.
INFO:Obtained after executing force state =OR(OR("bribe", "blackmail"), SAND(SAND(OK:OR("steal", OK:"force"), AND("di
INFO:Must =OR(OR("bribe", "blackmail"), SAND(SAND(OR("steal", "force"), REQ:AND("disableAlarm", "passDoor")), "takeDc
INFO:May =OR(OR(MAY:"bribe", MAY:"blackmail"), SAND(SAND(OR("steal", "force"), AND(MAY:"disableAlarm", MAY:"passDoor"
INFO:Starting recursion for action passDoor with 1 branches to explore.
INFO:Obtained after executing passDoor state =OR(OR("bribe", "blackmail"), SAND(SAND(OK:OR("steal", OK:"force"), AND(
INFO:Must =OR(OR("bribe", "blackmail"), SAND(SAND(OR("steal", "force"), REQ:AND("disableAlarm", "passDoor")), "takeDc
INFO:May =OR(OR(MAY:"bribe", MAY:"blackmail"), SAND(SAND(OR("steal", "force"), AND(MAY:"disableAlarm", "passDoor")),
INFO:Starting recursion for action disableAlarm with 1 branches to explore.
```

Fig. 11: Interface of the tool `https://github.com/yanntm/Abat`.

Below, we illustrate how our implementation works on an example. We specify the tree obtained from the reduction of the PIC instance considered on page 19 as follows:

```
tree = AND (
OR ( SAND ("a1","a2","a3","a4","a5","a6") , SAND("a5","a6","a7","a8","a9")),
OR ( SAND ("a1","a2","a3") , SAND("a4","a5","a6"), "a7" ),
"a4"
);
```

Then, we give the following traces:

```
trace interval_1_1 = "a1";
trace interval_1_2 = "a1","a2";
trace interval_1_3 = "a1","a2","a3";
trace interval_1_4 = "a1","a2","a3","a4";
trace interval_1_5 = "a1","a2","a3","a4","a5";
trace interval_1_6 = "a1","a2","a3","a4","a5","a6";
trace interval_1_7 = "a1","a2","a3","a4","a5","a6","a7";
trace interval_1_8 = "a1","a2","a3","a4","a5","a6","a7","a8";
trace interval_1_9 = "a1","a2","a3","a4","a5","a6","a7","a8","a9";
trace interval_1_10 = "a1","a2","a3","a4","a5","a6","a7","a8","a9","a10";
```

and the tool answers:

```
Trace "interval_1_1" rejected
Trace "interval_1_2" rejected
Trace "interval_1_3" rejected
Trace "interval_1_4" rejected
Trace "interval_1_5" rejected
Trace "interval_1_6" accepted
Trace "interval_1_7" accepted
Trace "interval_1_8" rejected
Trace "interval_1_9" accepted
Trace "interval_1_10" rejected
```

The current version of our implementation supports only trees without weak operators, but we will soon be able to consider arbitrary trees by extending the backtracking approach to the whole Algorithm 3.

6 Conclusion and future work

We have proposed a framework where attack trees are interpreted according to a model of the system, thus yielding their path semantics, but mostly displaying a natural notion of missing attacks. We then have considered the decision problem of the existence of a missing attack (MAE), which is highly pertinent for attack tree designers.

It should be noticed that our notion of missing attack relies on the model of the system and not on the system itself, just as the model checking principle in system verification makes the model design an upstream issue.

We insist on the robustness of the proposed approach for our models of systems: those are transition systems that allow for non-deterministic behavior. This way, the attacker executing an action may not control its effects, which captures the idea that the attacker interacts with some environment (seen as an abstract opponent that solves the non-determinism – a very standard way of modeling in formal methods). The attacks are thus sequences of actions that the attacker can entirely perform, if the environment does not prevent him from doing so.

Also, regarding our formal setting, we have equipped attack trees not only with standard operators, but also with weak variants of those, allowing more flexibility in the specification and getting much closer to our intuition when reading informal trees developed in practice.

From our path semantics, and by considering the universal system where any sequence of actions can be executed, we have also defined a trace semantics for attack trees, thus offering an interpretation of attack trees on their own, and exhibiting a formal notion to investigate the missing attack problem MAE. In particular, we studied the trace attack tree membership problem TATM and showed it is NP-complete. Noticeably, the NP-hardness proof for TATM, even for trees with no weak operators, resorts to a newly defined combinatorial problem PIC that, although very natural, has never been considered in the literature.

Next, relying on an NP oracle that answers TATM, we could design non-deterministic polynomial-time algorithm to solve MAE when weak operators are discarded. The algorithm guesses three paths and makes three independent calls to this oracle for each path, which shows its membership in the complexity class Σ_2^P of the polynomial hierarchy [24] (containing the classes NP and co-NP).

On our way to develop tools for attack tree users, we have implemented the NP oracles: on the basis of the non-deterministic algorithms described in Section 5, we have coded their deterministic version as a backtracking algorithm. The tool is freely available and open source. It is for now mostly dedicated to educational purposes for better understanding the chosen semantics of the operators.

There are several directions to pursue this work. First, complexity bounds need to be made tighter. While we know that MAE is in Σ_2^P and that it is as hard as any co-NP problem, we still need to fill this gap. Next, we should address the complexity of (full[5]) MAE, which requires to provide a bound on the three paths guessed by Algorithm 1 in the general case, rather than for trees with no weak operators.

Obviously the aforementioned missing complexity results would shed light on the difficulty of synthesizing missing attacks, or provide hints for subclasses of instances where the MAE problem might become simpler.

Moreover, it could be interesting to investigate cases where the problem TATM is easier. For example, when the tree does not contain any weak operators and actions appear at most once in the tree, the complexity of TATM becomes linear since there is only one way of interpreting an action in the tree.

Finally, we wish to study variants of the path/trace semantics of attack trees for weak operators. After all, weak operators offer a way to abstract from intermediate sequences of actions. These actions are neglected by the designer for some reason that need to be better understood. We may refine the current semantics by considering that these actions should be other than those occurring in the tree. The definition of the path semantics can be adapted accordingly, and does not change our current results, while giving a hope to obtain complexity upper bounds for MAE with arbitrary trees.

References

1. Amenaza: SecurITree (2001–2013), `http://www.amenaza.com/`
2. Audinot, M., Pinchinat, S., Kordy, B.: Is my attack tree correct? In: ESORICS. LNCS, vol. 10492, pp. 83–102. Springer (2017)
3. Audinot, M., Pinchinat, S., Kordy, B.: Guided design of attack trees: a system-based approach. In: CSF. IEEE Computer Society (2018)

[5] In the full MAE problem, all (strong and weak) operators are allowed.

4. Audinot, M., Pinchinat, S., Schwarzentruber, F., Wacheux, F.: Deciding the Non-Emptiness of Attack trees. In: GraMSec 2018. LNCS, vol. 11086, pp. 13–30. Springer (2018)
5. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
6. Berman, P., Karpinski, M., Scott, A.D.: Approximation hardness of short symmetric instances of MAX-3SAT. Electronic Colloquium on Computational Complexity (ECCC) (049) (2003), `http://eccc.hpi-web.de/eccc-reports/2003/TR03-049/index.html`
7. EAC Advisory Board and Standards Board: Election Operations Assessment – Threat Trees and Matrices and Threat Instance Risk Analyzer (TIRA) (2009), `https://www.eac.gov/assets/1/28/Election_Operations_Assessment_Threat_Trees_and_Matrices_and_Threat_Instance_Risk_Analyzer_(TIRA).pdf`
8. Gadyatskaya, O., Harpes, C., Mauw, S., Muller, C., Muller, S.: Bridging two worlds: Reconciling practical risk assessment methodologies with theory of attack trees. In: GraMSec 2016. LNCS, vol. 9987, pp. 80–93. Springer (2016)
9. Gadyatskaya, O., Jhawar, R., Mauw, S., Trujillo-Rasua, R., Willemse, T.A.C.: Refinement-Aware Generation of Attack Trees. In: STM. LNCS, vol. 10547, pp. 164–179. Springer (2017)
10. Hong, J.B., Kim, D.S., Chung, C., Huang, D.: A survey on the usability and practical applications of Graphical Security Models. Computer Science Review **26**, 1–16 (2017)
11. Isograph: AttackTree+ (2004–2005), `http://www.isograph-software.com/atpover.htm`
12. Ivanova, M.G., Probst, C.W., Hansen, R.R., Kammüller, F.: Attack Tree Generation by Policy Invalidation. In: WISTP. LNCS, vol. 9311, pp. 249–259. Springer (2015)
13. Jhawar, R., Kordy, B., Mauw, S., Radomirovic, S., Trujillo-Rasua, R.: Attack Trees with Sequential Conjunction. In: SEC. IFIP AICT, vol. 455, pp. 339–353. Springer (2015)
14. Jürgenson, A., Willemson, J.: Computing exact outcomes of multi-parameter attack trees. In: OTM Conferences (2). LNCS, vol. 5332, pp. 1036–1051. Springer (2008)
15. Kordy, B., Piètre-Cambacédès, L., Schweitzer, P.: DAG-based attack and defense modeling: Don't miss the forest for the attack trees. Computer Science Review **13-14**, 1–38 (2014)
16. Kordy, B., Wideł, W.: On quantitative analysis of attack–defense trees with repeated labels. In: POST. LNCS, vol. 10804, pp. 325–346. Springer (2018)
17. Mantel, H., Probst, C.W.: On the Meaning and Purpose of Attack Trees. In: CSF. IEEE Computer Society (2019)
18. Mauw, S., Oostdijk, M.: Foundations of Attack Trees. In: ICISC. LNCS, vol. 3935, pp. 186–198. Springer (2005)
19. National Electric Sector Cybersecurity Organization Resource (NESCOR): Analysis of selected electric sector high risk failure scenarios, version 2.0 (2015), `http://smartgrid.epri.com/doc/NESCOR%20Detailed%20Failure%20Scenarios%20v2.pdf`
20. Pinchinat, S., Acher, M., Vojtisek, D.: Towards Synthesis of Attack Trees for Supporting Computer-Aided Risk Analysis. In: SEFM Workshops. LNCS, vol. 8938, pp. 363–375. Springer (2014)

21. Pinchinat, S., Acher, M., Vojtisek, D.: ATSyRa: An Integrated Environment for Synthesizing Attack Trees – (Tool Paper). In: GraMSec. LNCS, vol. 9390, pp. 97–101. Springer (2015)
22. Saffidine, A., Cong, S.L., Pinchinat, S., Schwarzentruber, F.: The Packed Interval Covering Problem is NP-complete. CoRR **abs/1906.03676** (2019), `http://arxiv.org/abs/1906.03676`
23. Schneier, B.: Attack trees. Dr. Dobbs journal **24**(12), 21–29 (1999)
24. Stockmeyer, L.J.: The polynomial-time hierarchy. Theoretical Computer Science **3**(1), 1–22 (1976)
25. Vigo, R., Nielson, F., Nielson, H.R.: Automated Generation of Attack Trees. In: CSF. pp. 337–350. IEEE Computer Society (2014)