Library-based Attack Tree Synthesis

Sophie Pinchinat, François Schwarzentruber, and Sébastien Lê Cong

Univ Rennes/IRISA/CNRS

Abstract. We consider attack trees that can contain OR-, AND- and SAND-nodes. Relying on a formal notion of library inspired from context-free grammars, we introduce a generic attack tree synthesis problem that takes such a library and a trace as inputs. We show that this synthesis problem is NP-complete. The NP membership relies on an involved adaptation of the so-called CYK parsing algorithm, The NP hardness is established via a reduction from a recent covering problem. Finally, we show that the addressed synthesis problem collapses down to P for bounded-AND-arity libraries.

1 Introduction

In security analysis, *attack trees* [Sch99] offer a representation to describe many attacks with brevity. They offer a reading of high-level *explanations* of attacks using different levels of abstractions. Also, they are convenient to to perform quantitative analysis on attacks in order to select efficient counter-measures, as well as to identify attacker profiles. As general objects, they are useful in various situations in the industry: they are used for assessing the security of physical infrastructures [(NE15], cyber security platforms such as voting systems [BB09] or specific machines like an ATM [FFG+16], and also to conduct quantitative analyses of a system that uses radio-frequency identification (RFID) technology [BKMS12].

We here informally introduce the attack tree model on a toy running example in physical security.

Example 1. A museum has two possible entries, both monitored by the same two cameras. The two cameras have a mutual protection system (distinct from the visual surveillance) so that they monitor each other: if a camera gets frozen while being monitored by the other, then an alarm is triggered. In order to neutralize a camera, the attacker can launch a virus on any camera: this virus immediately disables its ability to monitor the other camera, then, possibly after some time, it freezes the camera. Additionally, the freezing is temporary so that a frozen camera may recover from freezing. The attack tree of Figure 1 describes ways of attacking the museum to steal the painting: each node of the tree matches a task, and the children of a node match the subtasks. This tree displays three types of inner nodes, that specify how the subtasks should be accomplished. In OR-nodes, one subtask has to be achieved. In SAND-nodes, subtasks should be realized sequentially (from left to right). In AND-nodes, all subtasks have to be executed in parallel. According to this tree, stealing the painting can be achieved for example by (1) turning the security off, then (2) entering the museum, and finally (3) taking the painting.

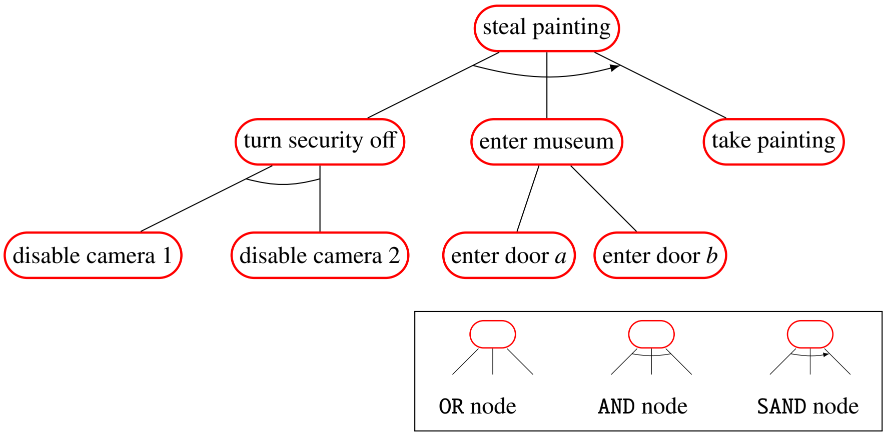

Fig. 1. An attack tree for stealing a painting in a museum with two doors, protected by two security cameras.

The design of attack trees faces a tedious and error-prone process if done manually: indeed, security experts may run into trouble as soon as the material they work on gets fairly big (lengthy log files, for example). In this context, gathering information becomes a complex task, and the resulting trees can get quite large. Hence, automated attack tree synthesis, even partial, is useful.

As shown in the related-work Section2, many algorithms have been proposed for several variants of attack tree synthesis. In particular, some previous works rely on models for representing the accumulated expert knowledge about existing attack patterns, in order to synthesize attack trees that speak to experts [JLMRC18,GJM+17]. Regrettably, the quality of the deployed algorithms can hardly be evaluated because of a lack of results on the intrinsic complexity of the tree synthesis problem.

It is therefore desirable to have a clear understanding of the attack tree synthesis problem(s) at a theoretical level in order to justify any algorithm. This requires a sleek definition of the attack tree synthesis problem, generic and simple enough to capture the core difficulty of the issue.

The present paper is about such a study. Our mathematical setting is the one of attack trees with a trace semantics, in the spirit of [APK17,Aud18]. The main reason for it comes from the genericity of the notion of trace. Indeed, traces can be found in most domains: as abstractions of system executions in verification, as sequences of events in monitoring, as log files in security, as plans in AI, as sequences of letters in formal languages and in bioinformatics, etc.

We define the notion of *library* as an abstract model for some expert knowledge, inspired from context-free grammars [HMU07], and generic enough to resemble proposals from the literature on attack tree generation, and in particular the ones of [JLMRC18] and [GJM+17].

Importantly, our approach is *model-free* which makes it relevant for situations where the system model is unknown; only a trace, reminiscent of some system observation, matters.

The synthesis decision problem, that we simply call the *attack tree synthesis problem* is defined as: given an input a *library* and an input *trace*, answers whether there exists an attack tree based on the given library whose trace semantics contains the input trace.

We prove that the attack tree synthesis problem is NP-complete. Noticeably, its NP-hardness is obtained by reducing the recently considered "Packed Interval Covering Problem" [SCPS19]. The NP-membership relies on a non-trivial adaptation of the classic Cocke–Younger–Kasami parsing algorithm [Kas66]. Interestingly, we highlight the role of the `AND`-operator by showing a drop to the class P in the problem complexity if the arity of this operator is bounded in the input libraries.

The paper is organized as follows: in Section 2, we consider related works and their limits. In Sections 3 and 4, we settle the formal setting of attack trees with their trace semantics and with the library model, respectively. Section 5 contains the full synthesis problem study. The paper ends with a concluding section and research perspectives.

2 Related Work

We focus on the attack tree synthesis literature for the last two decades, in a chronological order; the reader interested in a survey on attack tree literature can refer to [WAFP19] (notice that the assumptions are quite diverse, but that there is an agreement that attack trees should help experts reasoning about ways of attacking a system). In some contributions, the formal semantics of attack trees is omitted, which makes hard stating properties of the generated trees, and in particular about what they desbribe. Also some works do not define the synthesis problem as a formal problem, making hard to evaluate the efficiency of the proposed approach with regards to the intrinsic complexity of the problem.

In Hong et al. [HKT13], the semantics of the considered attack trees is not provided. The tree generation does not rely on any notion of library. The input is a set of attacks (that can be given or inferred as paths from some attack graph). Their procedure considers as the first step the naive tree obtained as the complete disjunction of all input attacks, where each attack is represented by the mere sequential conjunction of all its actions. In a second step, (although not told this way in the paper) the procedure resorts to controlled regular expression manipulations to make the former huge tree hopefully smaller. The purpose of this technique is mostly used to achieve quantitative analysis in an attack graph, and does not target readability of the tree. No meaning of the subtasks that inhabit the internal sub-nodes can be inferred by this procedure that artificially creates internal nodes from algebraic laws on regular expressions. Also, the approach lacks the use of `AND` operator that can provide more succinct trees and indeed, as explained by the authors, the synthesized trees have exponential size in the size of the input.

Vigo et al. [VNN14] do not use a library and do not consider the sequential conjonction of subtasks (`SAND` operator). The input are a "program" representing the system and a point to reach in the former. The programs are described in so-called "value-passing quality calculus", a calculus which derives from the π-calculus. The system program with its point to reach is translated into a propositional formula that is interpreted as an attack tree (with intended meaning of disjunction and conjunction operators). However, since the internal nodes of the synthesized trees are abstract, the resulting trees are used more for quantitative analysis than for explaining ways of attacking.

Pinchinat et al. [PAV14,PAV15] present a tool for synthesizing attack trees. The method is very close to our approach, since it is based on a library, and on a bottom-up construction of the tree inspired from context-free grammar syntactic analysis. The used library is defined aside the synthesis functionality; it can be defined manually in the tool, but may also be imported from previous projects. However, the procedure does not support operator `AND`.

In the setting of Ivanova et al. [IPHK15], the authors suggest a high-level language intended to turn a graph, a so-called "graphical system model", into an attack tree with the intention to make the graph more readable. Those graphical models specify an initial state of some system – vertices represent elements (such as doors, agents, information, and so on), and the attacker has to reach some final configuration. The translation from one setting to another does not rely on a precise semantic framework. The translation from the graph to an attack tree is generic, not taking advantage of any specific expert knowledge. The library is implicitly based on ad-hoc patterns (with first-order logic features) correlated with fixed ontologies (locations, actors, processes, items). As a result, the obtained trees are unbalanced, and not readable. Also, only disjunction and sequential conjunction are considered.

Gadyatskaya et al. [GJM$^+$17] define a library-based generic synthesis problem parameterized by the semantics of attack trees. The library is called a refinement specification. However, the paper focuses on the particular serie-parallel graph (SP) semantics, where the `AND` operator has a truly-concurrent meaning. Surprisingly, the authors restrict to SP graphs without any `AND` operator, that is as a set of traces. This prevents to address the synthesis problem for arbitrary refinement rules. Also, the paper does not provide the complexity analysis of the addressed synthesis problem. The tree models we consider here are not based on actions (at the leaves), but it can be established that our semantics semantics coincides with the SP semantics if the `AND` operator is discarded. Our synthesis problem can therefore be seen as a restriction of their work to a singleton set of single traces, but also as an extension of it as we allow one to `AND` operator.

Jhawar et al. [JLMRC18] consider the issue of automating the completion of an attack tree rather than synthesizing one, by an iterated top-down approach. A criterion based on annotations of nodes with preconditions and postconditions, makes it possible to attach subtrees from some library at some leaves. The logical setting to describe the annotations lacks dynamic features (such as temporal modalities) amenable to the use sequential conjunction.

In [APSW18], Audinot et al. study the non-emptiness of an attack tree, in a framework similar to what we consider here: given an attack tree, they query the existence of an attack described by the input tree. Our problem can be read as the dual of this problem since the trace is known but the tree has to be found.

3 Attack Trees and Their Trace Semantics

We consider the setting of [APK17], where attack tree leaves are labeled by atomic goals, but due to our concern, we equip them with a trace semantics instead of a path semantics, in natural manner. Indeed, traces are mere abstraction of finite paths (in some transition system), by replacing each state along the path by its set of true facts; thus a trace is a finite sequence of facts. In formal approaches facts are modeled by abstract *propositions* in a set $Prop = \{p, q, r, \dots\}$.

An atomic goal at a leaf of an attack tree is composed of a *precondition* and a *postcondition*, and denotes the set of finite sequences of true propositions where the precondition holds at their beginning and the postcondition holds at their end. The *trace semantics* of a non-leaf attack tree is given in a compositional manner by means of operations on (sets of) finite sequences, such as concatenation.

We now get into the formal definitions.

3.1 Attack Trees

Formally, an attack tree is a tree whose leaves are *atomic goals* of the form $\langle \iota \textbf{ to } \gamma \rangle$, where ι and γ are Boolean formulae over a set of atomic propositions *Prop*, called the *precondition* and the *postcondiion* respectively. Each inner node of an attack tree is labelled by some operator `OP` ranging over `OR` (disjunction), `SAND` (sequential conjunction) or `AND` (conjunction), and is called an *`OP`-node*.

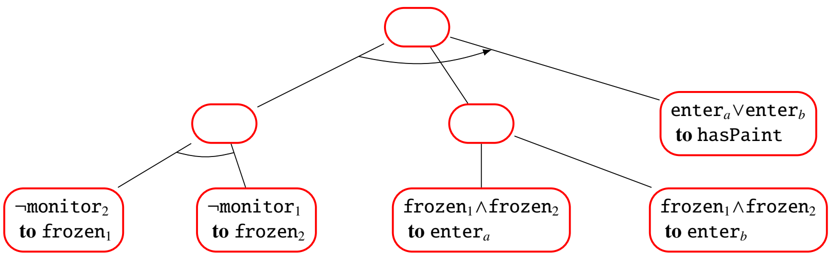

Fig. 2. The formal attack tree for the museum example.

Example 2. Figure 2 shows a formalisation of the informal attack tree from Figure 1, with 3 inner nodes and 5 leaves. Propositions occurring in the atomic goals of the leaves are interpreted as follows: $\texttt{monitor}_i$ means "camera i is being monitored (by the other camera)", $\texttt{frozen}_i$ means "camera i is frozen", $\texttt{enter}_j$ means "entered in museum via door j", and `hasPaint` means "the painting was stolen". Therefore, the atomic goal $\langle \neg\texttt{monitor}_2 \textbf{ to } \texttt{frozen}_1 \rangle$ models the task of hacking camera 1: launching the virus immediately stops camera 1 from monitoring camera 2 and eventually freezes camera 1. Symmetrically, goal $\langle \neg\texttt{monitor}_1 \textbf{ to } \texttt{frozen}_2 \rangle$ regards the hacking of camera 2. We will elaborate on the camera-hacking phase later, in Subsection 3.4. Also, goal $\langle \texttt{frozen}_1 \wedge \texttt{frozen}_2 \textbf{ to } \texttt{enter}_a \rangle$ models the task of entering the museum via door a without surveillance.

Definition 1 (Attack tree). *An* attack tree τ *(over Prop) is:*

- *either a leaf of the form* $\langle \iota \textbf{ to } \gamma \rangle$ *where* ι, γ *are Boolean formulae over Prop;*
- *or a construction* $\texttt{OP}(\tau_1, \ldots, \tau_m)$ *where* $\texttt{OP}$ *is the operator* $\texttt{OR}$, $\texttt{AND}$ *or* $\texttt{SAND}$, $m \geq 1$ *is the arity, and* $\tau_1, \ldots, \tau_m$ *are attack trees.*

In Definition 1 we confuse a node and the subtree rooted at that node. This is standard when trees are defined inductively.

Example 3. The attack tree given in Figure 2 is

$$\begin{array}{l}\texttt{SAND}(\ \texttt{AND}(\langle \neg\texttt{monitor}_2 \textbf{ to } \texttt{frozen}_1 \rangle, \langle \neg\texttt{monitor}_1 \textbf{ to } \texttt{frozen}_2 \rangle), \\ \quad \texttt{OR}(\langle \texttt{frozen}_1 \wedge \texttt{frozen}_2 \textbf{ to } \texttt{enter}_a \rangle, \\ \quad\quad \langle \texttt{frozen}_1 \wedge \texttt{frozen}_2 \textbf{ to } \texttt{enter}_b \rangle), \\ \quad\quad \langle \texttt{enter}_a \vee \texttt{enter}_b \textbf{ to } \texttt{hasPaint} \rangle)\end{array}$$

The second central objects of concern are *traces*.

3.2 Traces and operations on sets of traces

Executions of systems are alternating sequences consisting of states and actions. In our setting for attack trees, the focus is put on states. In fact, the states themselves are not "observable" along an execution, but only the truth value of facts/propositions about them. A truth value of propositions is formally captured by the standard notion of *valuation* in propositional logic. Thus an observation of a (finite) execution, usually called a *trace* [BK08], is a finite sequence of valuations; two successive valuations in a trace correspond to a state transition in the observed system.

We now formally define *traces*, sets of traces, and particular operations over languages that provide the semantics of operators $\texttt{OR}$, $\texttt{SAND}$ and $\texttt{AND}$ in attack trees.

For the rest of this section, we fix a set *Prop* of propositions.

A *valuation* is a subset of *Prop* with the meaning that propositions in this set are true while the others are false; for the empty valuation $\emptyset$, all propositions are thus false. We therefore write 2^{Prop} for the set of valuations on the set *Prop*, with typical element $\nu \in 2^{Prop}$. Given a Boolean formula φ over *Prop*, we write $\nu \models \varphi$ to denote that ν satisfies φ.

Traces are finite sequences of valuations, and we denote by ε for the empty sequence. Given a trace $t \in (2^{Prop})^*$, the *length* $|t|$ of t is defined as its number of valuations. For $1 \leq i \leq |t|$, the i^{th} valuation of t is denoted by $t(i)$. We set $t.\mathit{first} = t(1)$ and $t.\mathit{last} = t(|t|)$ and we denote by $t[i, j]$ the subsequence of t starting at position i and ending at position j. For instance, if $t = \nu_1\nu_2\nu_3\nu_4\nu_5$, then $t.\mathit{first} = \nu_1$, $t.\mathit{last} = \nu_5$ and $t[2, 4] = \nu_2\nu_3\nu_4$.

Example 4. Consider $\{\texttt{monitor}_1\}$ $\{\texttt{monitor}_1\}$ $\emptyset$ $\{\texttt{frozen}_1\}$ $\{\texttt{frozen}_1, \texttt{frozen}_2\}$ $\{\texttt{enter}_b, \texttt{frozen}_1, \texttt{frozen}_2\}$ $\{\texttt{hasPaint}, \texttt{frozen}_1, \texttt{frozen}_2\}$ a trace of length 7 from the museum example. It reflects the scenario where, during the first two timesteps, both cameras work, camera 1 is monitored and camera 2 is not. At the third step, camera 1 is not any more monitored. Then, camera 1 is frozen, before camera 2. Next, the intruder enters the building via door b while both cameras are frozen, and finally steals the painting while the cameras are still frozen.

In the following, we may write traces with arrows between their valuations in order to emphasize the underlying state transitions that take place: $t = v_1 \to v_2 \to v_3 \to v_4 \to v_5$.

Regarding the trace semantics of attack trees that will be given in Definition 4, the OR operator will be understood as the union operation over sets of traces, whereas the two other operators SAND and AND will be given less classic interpretations that we present now.

3.3 Synchronized concatenation

The *synchronized concatenation* $\odot$ slightly differs from the usual concatenation in formal languages and conveys the notion of sequential executions of scenarios; it will provide the semantics of the SAND operator in attack trees.

Definition 2 (Synchronized concatenation).
The synchronized concatenation *of two traces is defined only if the last valuation of the former is equal to the first valuation of the latter, and simply concatenates the two traces by merging this common element. Formally,*

$$v_1 \dots v_n v \odot v v'_1 v'_2 \dots v'_m = v_1 \dots v_n v v'_1 \dots v'_m.$$

Example 5. $\{\texttt{frozen}_1, \texttt{frozen}_2\}\{\texttt{enter}_b, \texttt{frozen}_1, \texttt{frozen}_2\} \odot \{\texttt{enter}_b, \texttt{frozen}_1, \texttt{frozen}_2\}$ $\{\texttt{hasPaint}, \texttt{frozen}_1, \texttt{frozen}_2\}$ $=$ $\{\texttt{frozen}_1, \texttt{frozen}_2\}$ $\{\texttt{enter}_b, \texttt{frozen}_1, \texttt{frozen}_2\}$ $\{\texttt{hasPaint}, \texttt{frozen}_1, \texttt{frozen}_2\}$; the synchronized concatenation is possible thanks to the common matching valuation $\{\texttt{enter}_b, \texttt{frozen}_1, \texttt{frozen}_2\}$.

The synchronized concatenation $\odot$ is associative, so that binary $\odot$ suffices. We lift the synchronized concatenation to sets L, L' of traces by letting

$$L \odot L' = \{t \odot t' \mid t \in L, t' \in L' \text{ and } t \odot t' \text{ is defined}\}.$$

3.4 Parallel composition

The *parallel composition* written $\bowtie$ is adapted from [APK17] to traces. This operation reflects the meaning of achieving subgoals in a concurrent manner, and aims at capturing what the AND operator expresses in attack trees. We motivate its definition on an example with the concurrent achievement of two atomic goals: consider the AND-node from Figure 2 and the following trace (a prefix of the trace in Example 4) realizing a successful hacking of both cameras, namely goal $\langle\neg\mathtt{monitor}_2 \textbf{ to } \mathtt{frozen}_1\rangle$ and goal $\langle\neg\mathtt{monitor}_1 \textbf{ to } \mathtt{frozen}_2\rangle$.

$$\underbrace{\{\mathtt{monitor}_1\} \rightarrow \{\mathtt{monitor}_1\} \rightarrow \mathrlap{\overbrace{\phantom{\emptyset \rightarrow \{\mathtt{frozen}_1\} \rightarrow \{\mathtt{frozen}_1, \mathtt{frozen}_2\}}}^{\langle\neg\mathtt{monitor}_1 \textbf{ to } \mathtt{frozen}_2\rangle}}\emptyset \rightarrow \{\mathtt{frozen}_1\}}_{\langle\neg\mathtt{monitor}_2 \textbf{ to } \mathtt{frozen}_1\rangle} \rightarrow \{\mathtt{frozen}_1, \mathtt{frozen}_2\} \tag{1}$$

Right from the start, camera 1 gets a virus and cannot monitor camera 2 ($\mathtt{monitor}_2$ is false). The observation does not change for one step, and then, camera 2 gets infected too ($\mathtt{monitor}_1$ turns false). Then, camera 1 gets frozen first ($\mathtt{frozen}_1$), and next camera 2 does too ($\mathtt{frozen}_2$). Realizing the conjunction of the hacking subgoals means that they are executed concurrently: any transition of the global hacking task falls under one of the hacking subgoals, and the global task is embedded in the achievement of both subgoals. On the contrary, the following trace does not reflect a conjunction of the two hacking subgoals because the second transition does not serve any of the hacking subgoals.

$$\underbrace{\{\mathtt{monitor}_1\} \rightarrow \{\mathtt{monitor}_1, \mathtt{frozen}_1\}}_{\langle\neg\mathtt{monitor}_2 \textbf{ to } \mathtt{frozen}_1\rangle} \rightarrow \underbrace{\{\mathtt{monitor}_2\} \rightarrow \{\mathtt{monitor}_2, \mathtt{frozen}_2\}}_{\langle\neg\mathtt{monitor}_1 \textbf{ to } \mathtt{frozen}_2\rangle}$$

In concrete terms, a virus is launched on camera 1, then camera 1 gets frozen, then a virus is launched on camera 2 while camera 1 gets back to normal operation, then finally, camera 2 gets frozen. In this scenario, the second hacking task starts too late and the alarm is triggered (camera 1 is able to notice the discrepancy in camera 2's behaviour). The `AND`-node of the tree expresses that it is necessary for the two hacking subgoals to take place with some overlapping of their transitions to be successful. This is formalized in Definition 3 as *parallel composition* of traces which can be interpreted as follows: if one sees a trace, of length n, as displaying some "activity", every transition (*i.e.,* action) along this trace corresponds to a 1-length subinterval $[k, k+1] \subseteq [1, n]$, while subgoals correspond to arbitrary subintervals. In the example, the camera 1 hacking subgoal of the 5-length trace of Expression (1) corresponds to subinterval $[1, 4]$ and the camera 2 hacking subgoal corresponds to subinterval $[3, 5]$. Therefore each transition along this trace serves at least one of the two camera hacking subgoals.

More formally, let us say that the intervals $I_1, \ldots, I_m$ *cover* an interval I whenever

$$\bigcup_{\ell=1}^{m} I_\ell = I \text{ and each } [k, k+1] \subseteq I \text{ is contained in some } I_\ell.$$

We can now proceed to the formal definition of the parallel composition.

Definition 3 (Parallel composition). *A trace t is a* parallel composition *of traces $t_1, \ldots t_m$ if there are m intervals $I_1, \ldots, I_m$ that cover $[1, n]$ and such that $t[I_\ell] = t_\ell$, for every $1 \leq \ell \leq m$. We also simply say that traces $t_1, \ldots t_m$* cover *trace t.*

Example 6. Figure 3 shows that the trace $t = v_1 \ldots v_7$ is a parallel composition of traces t_1, t_2 and t_3 with respective intervals $[1, 2]$, $[4, 7]$, $[2, 5]$. Indeed, all transitions $v_1 \rightarrow v_2$, $v_2 \rightarrow v_3, \ldots, v_6 \rightarrow v_7$ are covered. On the contrary, t is not a parallel composition of t_1, t_2 and t'_3 since the only interval candidates are respectively $[1, 2]$, $[4, 7]$, $[3, 5]$, but none of them fully includes the subinterval $[2, 3]$. In other words, the transition $v_2 \rightarrow v_3$ is not covered.

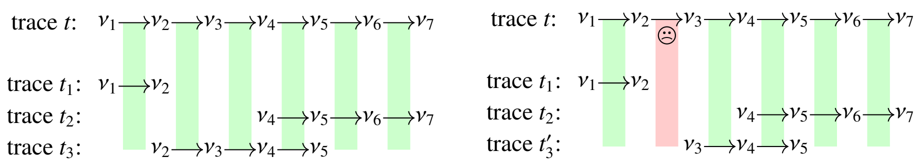

Fig. 3. The trace t is a parallel composition of t_1, t_2, t_3 but not of t_1, t_2, t_3'.

The parallel composition reflects the conjunctive execution of activities and not the conjunction of the effects of these activities, which is a legitimate interpretation of the AND operator in attack trees (see the serie-parallel graph semantics considered in [GJM+17]). Typically, requiring to open and to close a door does mean to attain a situation where the door is both open and closed.

Traces $t_1, \ldots t_m$ may cover several traces, i.e. may have several parallel compositions. We let $\mathbin{/\!\!\!\wedge\!\!\!\backslash}(t_1, \ldots, t_m)$ be the set of parallel compositions of $t_1, \ldots t_m$.

Example 7. $\mathbin{/\!\!\!\wedge\!\!\!\backslash}(v'vv, vvv'') = \{v'vvv'', v'vvvv''\}$.

We lift the parallel composition to sets $L_1, \ldots, L_m$ of traces by letting

$$\mathbin{/\!\!\!\wedge\!\!\!\backslash}(L_1, \ldots, L_m) = \bigcup_{t_1 \in L_1, \ldots, t_m \in L_m} \mathbin{/\!\!\!\wedge\!\!\!\backslash}(t_1, \ldots, t_m).$$

It should be remarked that the synchronized concatenation $\odot$ is associative, so that binary $\odot$ suffices, while this is not the case of $\mathbin{/\!\!\!\wedge\!\!\!\backslash}$ in general: for example, $v_1v_2v_3v_4 \in \mathbin{/\!\!\!\wedge\!\!\!\backslash}(v_1v_2, v_3v_4, v_2v_3)$, but $\mathbin{/\!\!\!\wedge\!\!\!\backslash}(\mathbin{/\!\!\!\wedge\!\!\!\backslash}(v_1v_2, v_3v_4), v_2v_3) = \emptyset$ because v_1v_2 and v_3v_4 do not share any valuation.

3.5 Trace semantics of attack trees

Now, we define the *trace semantics* of attack trees. Operators in attack trees are interpreted as operations on trace sets: `OR` means union $\cup$, `SAND` means synchronized concatenation $\odot$, and `AND` mean parallel composition $\mathbin{/\!\!\!\wedge\!\!\!\backslash}$.

Definition 4 (Trace semantics of attack tree). *The* trace semantics *of an attack tree τ is a set of traces $L(\tau) \subseteq (2^{Prop})^*$, inductively defined on τ:*

$$\begin{aligned}
L(\langle \iota \textbf{ to } \gamma \rangle) &= \{t \in (2^{Prop})^* \mid t.first \models \iota \text{ and } t.last \models \gamma\}; \\
L(OR(\tau_1, \ldots, \tau_m)) &= L(\tau_1) \cup \ldots \cup L(\tau_m); \\
L(SAND(\tau_1, \ldots, \tau_m)) &= L(\tau_1) \odot \ldots \odot L(\tau_m); \\
L(AND(\tau_1, \ldots, \tau_m)) &= \mathbin{/\!\!\!\wedge\!\!\!\backslash}(L(\tau_1), \ldots, L(\tau_m)).
\end{aligned}$$

Since the `SAND` operator relies on the associative operation $\odot$, we may sometimes assume for convenience and w.l.o.g. that the degree of the `SAND`-nodes is 2. In contrast, such an assumption would not hold for operator `AND` since to $\mathbin{/\!\!\!\wedge\!\!\!\backslash}$ is not associative.

Example 8. Revisiting the attack tree τ from Example 3, the following trace from Example 4 is a possible trace of the musuem example that can be explained by the tree τ, *i.e.,* that is in $L(\tau)$:

$$\{\mathtt{monitor}_1\}\{\mathtt{monitor}_1\}\varnothing\{\mathtt{frozen}_1\}\{\mathtt{frozen}_1,\mathtt{frozen}_2\}\{\mathtt{enter}_b,\mathtt{frozen}_1,\mathtt{frozen}_2\}$$
$$\{\mathtt{hasPaint},\mathtt{frozen}_1,\mathtt{frozen}_2\}.$$

Indeed, first its prefix $\{\mathtt{monitor}_1\}\{\mathtt{monitor}_1\}\varnothing\{\mathtt{frozen}_1\}\{\mathtt{frozen}_1,\mathtt{frozen}_2\}$ belongs to $L(\mathtt{AND}(\langle\neg\mathtt{monitor}_2\ \textbf{to}\ \mathtt{frozen}_1\rangle,\langle\neg\mathtt{monitor}_1\ \textbf{to}\ \mathtt{frozen}_2\rangle))$, as a parallel composition of $\{\mathtt{monitor}_1\}\{\mathtt{monitor}_1\}\varnothing\{\mathtt{frozen}_1\} \in L(\langle\neg\mathtt{monitor}_2\ \textbf{to}\ \mathtt{frozen}_1\rangle)$ and $\varnothing\{\mathtt{frozen}_1\}\{\mathtt{frozen}_1,\mathtt{frozen}_2\} \in L(\langle\neg\mathtt{monitor}_1\ \textbf{to}\ \mathtt{frozen}_2\rangle)$. Second, its factor $\{\mathtt{enter}_b,\mathtt{frozen}_1,\mathtt{frozen}_2\} \in L(\langle\mathtt{frozen}_1\wedge\mathtt{frozen}_2\ \textbf{to}\ \mathtt{enter}_a\rangle)$, thus its belongs to the trace semantics of the subtree of τ rooted at the OR-node. Third, its suffix $\{\mathtt{enter}_b,\mathtt{frozen}_1,\mathtt{frozen}_2\}\{\mathtt{hasPaint},\mathtt{frozen}_1,\mathtt{frozen}_2\}$ achieves the last subgoal goal of SAND-node root of τ.

4 Libraries

The attack tree synthesis problem seems trivial: the single-node tree $\langle\top\ \textbf{to}\ \top\rangle$, where formula $\top$ means tautologically true, explains any trace! In order to synthetize interesting attack trees, we consider a *library*, that is a set of *refinement rules*, alike a context-free grammar rules. We will as much as possible keep close to notations introduced in [GJM+17]: for instance, we use ρ to denote a refinement rule.

In a context-free grammar style, we consider $\mathcal{G}$ a finite set of *non-terminal goals*, with typical elements g, g_1, g_2, and *terminal goals* that are atomic goals $\langle\iota\ \textbf{to}\ \gamma\rangle$ (where ι,γ are Boolean formulae).

Definition 5 (Refinement rules and library). *A* refinement rule *(over $\mathcal{G}$) ρ is either a so-called* elementary rule *$g \lhd \langle\iota\ \textbf{to}\ \gamma\rangle$ where ι,γ are Boolean formulae; or a rule $g \lhd \mathtt{OP}(g_1,\ldots,g_m)$ where OP is an operator, $m \geq 1$, and $g_1,\ldots,g_m \in \mathcal{G}$.*
A refinement rule $g \lhd \mathtt{OP}(g_1,\ldots,g_m)$ refines *g.*

The arity *of a refinement rule is 0 if it is elementary, and the arity of the operator OP appearing in the rule otherwise.*

A library *$\mathcal{L}$ over $\mathcal{G}$ is a finite set of refinement rules (over $\mathcal{G}$). The* size *of $\mathcal{L}$ is the total number of non-terminal goal occurrences that appear in all its rules, both in left-hand and right-hand sides of rules.*

Example 9. Let us continue with the museum example where we add the proposition incenter read as "the intruder is in the control center". The following set of rules $\mathcal{L}_{\text{museum}}$ is library (and relies on the vocabulary of Example 2), where non-terminal goals are sentences written in italic to emphazise their role in our model of a library.

$$\begin{cases} \textit{go to center} & \lhd \langle \top \textbf{ to } \mathtt{incenter} \rangle \\ \textit{blow up a bomb} & \lhd \langle \mathtt{incenter} \textbf{ to } \mathtt{frozen}_1 \wedge \mathtt{frozen}_2 \rangle \\ \textit{enter via door a} & \lhd \langle \mathtt{frozen}_1 \wedge \mathtt{frozen}_2 \textbf{ to } \mathtt{enter}_a \rangle \\ \textit{enter via door b} & \lhd \langle \mathtt{frozen}_1 \wedge \mathtt{frozen}_2 \textbf{ to } \mathtt{enter}_b \rangle \\ \textit{take} & \lhd \langle \mathtt{enter}_a \vee \mathtt{enter}_b \textbf{ to } \mathtt{hasPaint} \rangle \\ \textit{disable camera } 1 & \lhd \langle \neg \mathtt{monitor}_2 \textbf{ to } \mathtt{frozen}_1 \rangle \\ \textit{disable camera } 2 & \lhd \langle \neg \mathtt{monitor}_1 \textbf{ to } \mathtt{frozen}_2 \rangle \\ \textit{steal} & \lhd \mathtt{SAND}(\textit{disable camera }, \textit{enter}, \textit{take}) \\ \textit{disable cameras} & \lhd \mathtt{AND}(\textit{disable camera } 1, \textit{disable camera } 2) \\ \textit{disable cameras} & \lhd \mathtt{SAND}(\textit{go to center}, \textit{blow up a bomb}) \\ \textit{enter} & \lhd \mathtt{OR}(\textit{enter via door a}, \textit{enter via door b}) \end{cases}$$

Goal *go to center* represents reaching the control center (without any precondition, which is written $\top$), while goal *blow up a bomb* represents setting up a bomb that will disable both cameras while being in the control center. The other goals are clear. Note that there are two rules that refine goal *disable cameras* which reflects different ways of disabling both cameras. Allowing for different refinement rules for an abstract goal is of utter importance because libraries are filled by experts analysing different systems: for example, the rule to hack a USB key may drastically vary depending on the underlying OS. Encapsulating alternatives into a single OR mean that they may occur in the same system. Having a different rule for each alternative means that they correspond to different systems.

We now fix a library $\mathcal{L}$ over some set of non-terminal goals $\mathcal{G}$. We define $\mathcal{L}$-attack trees, in the spirit of what was called a "correct tree" in [GJM+17]: intuitively, they are attack trees obtained by iteratively applying refinement rules of the library on leaf-nodes until the leaves correspond to atomic goals.

Definition 6 ($\mathcal{L}$-attack tree). *An* $\mathcal{L}$-attack tree *is an attack tree τ (in the sense of Definition 1) equipped with a mapping ℓ that maps every node of τ onto a non-terminal goal of $\mathcal{G}$ in such a way that:*

- *if x is a leaf $\langle \iota$ **to** $\gamma \rangle$, then the rule $\ell(x) \lhd \langle \iota$ **to** $\gamma \rangle$ is in $\mathcal{L}$;*
- *if x is a node $\mathtt{OP}(x_1, \ldots, x_k)$ then the rule $\ell(x) \lhd \mathtt{OP}(\ell(x_1), \ldots, \ell(x_k))$ is in $\mathcal{L}$.*

The label $\ell(x)$ of a node in Definition 6 is a non-terminal goal. This non-terminal goal arising from the librairy carries information, such a text – as done in Example 9, or a CVE identifier[1]. It is this information that makes $\mathcal{L}$-attack trees readable to experts.

Example 10. Figure 4 shows two $\mathcal{L}_{\text{museum}}$-attack trees for $\mathcal{L}_{\text{museum}}$ defined in Example 9.

We say that the non-terminal goal g *derives* the trace t if there exists an $\mathcal{L}$-attack tree τ whose root's label is g and such that t is in $L(\tau)$.

Given a library $\mathcal{L}$, we can always manage to find an equivalent library $\mathcal{L}'$ where all SANDs are binary, in the sense that the trace semantics an $\mathcal{L}'$-attack tree is equal to

[1] CVE is a dictionary of publicly disclosed cybersecurity vulnerabilities and exposures https://cve.mitre.org/cve/.

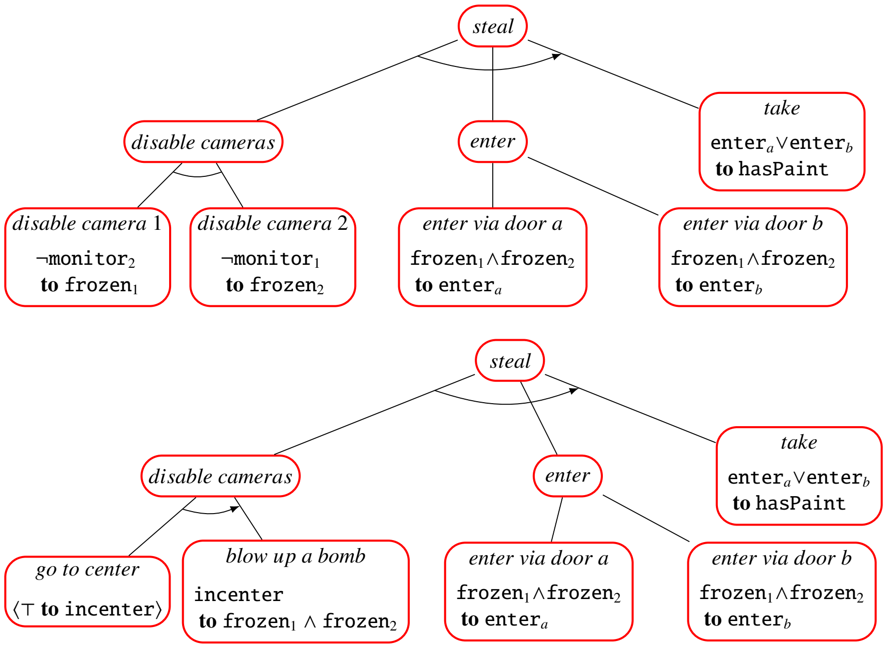

Fig. 4. Two $\mathcal{L}_{\text{museum}}$-attack trees.

trace semantics of some $\mathcal{L}$-attack trees, and vice versa. Note that $\mathcal{L}'$ can be computed in polynomial time in the size of $\mathcal{L}$.

In the rest of this paper, we assume that every refinement rule based on `SAND` operator has arity 2.

Table 1 sums up the formal notions defined so far.

Formal notions	**Intuitive meanings**
a trace	an observed attack (e.g. a log file)
an attack tree (Def. 1)	an explanation of an observed attack
a non-terminal goal	a high-level attack objective
a refinement rule	a known attack tree pattern
a library (Def. 5)	a set of known attack tree patterns
an $\mathcal{L}$-attack tree (Def. 6)	an explanation of an observed attack constructed with the known attack-tree patterns in $\mathcal{L}$

Table 1. Important formal notions defined in the paper.

5 Attack Tree Synthesis

The attack tree synthesis problem consists in building a tree (if any) that *explains* an observed trace t (e.g. a log file) in terms of a given library $\mathcal{L}$. Formlly, we address the underlying decision problem for analyzing the complexity for this synthesis problem, but the developped algorithm does build a tree.

Definition 7 (Attack tree synthesis problem).

- *Input: a library* $\mathcal{L}$*, a trace* $t \in (2^{Prop})^*$.
- *Output: is there an* $\mathcal{L}$*-attack tree* τ *such that* $t \in L(\tau)$*?*

The rest of this section is dedicated to the proof of the following theorem.

Theorem 1. *The attack tree synthesis problem is* NP*-complete. Furthermore, the synthesis problem restricted to libraries in which the arity of* `AND` *is bounded is in* P.

For proving the NP-hardness of the attack tree synthesis problem, we identify a decision problem as the core of the synthesis problem: the "Packed Interval Covering Problem" [SCPS19].

5.1 A detour on the Packed Interval Covering Problem

The Packed Interval Covering Problem (PIC) is a cover problem, where one has to cover a given interval using one interval from each given pack. It is defined as follows.

- Input: a non-empty interval I of integers and a family of finite sets $P_1, \ldots, P_m$ (*packs*) of subintervals of I.
- Output: are there subintervals $I_1 \in P_1, \ldots, I_m \in P_m$ such that $I = \bigcup_{k=1..m} I_k$?

Example 11. We borrow the example in [SCPS19]: for interval $[1, 9]$, there are three packs $\{[1, 6], [5, 9]\}$, $\{[1, 3], [4, 6], [7, 7]\}$, $\{[4, 4]\}$. Interval $[1, 9]$ can be covered by selecting $[5, 9]$, $[1, 3]$ and $[4, 4]$ in the respective packs, as shown in Figure 5.

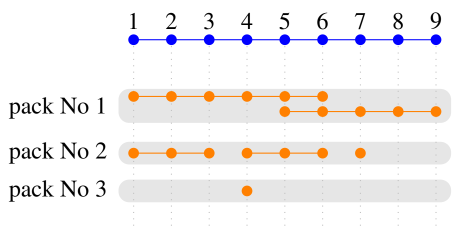

Fig. 5. Example of an instance of the Packed Interval Covering Problem.

Theorem 2 ([SCPS19]). PIC *is* NP*-complete.*

Proof. See Appendix A.

5.2 NP-hardness of the synthesis problem

We establish a reduction from PIC to the attack tree synthesis problem.

Consider an arbitrary instance of PIC with target interval $I = [1, N]$ and packs $(P_k)_{1 \leq k \leq m}$, each of the form $P_k = \{[m^k_j, n^k_j] \mid 1 \leq j \leq |P_k|\}$.

We now describe an instance $\langle \mathcal{L}, t \rangle$ of the attack tree synthesis problem as follows. Take N distinct propositions $p_0, \ldots, p_N$.

First, define trace $t = \{p_0\} \ldots \{p_N\}$ to encode the target interval $[1, N]$: each subtrace $\{p_{i-1}\}\{p_i\}$ of t of length 2 is intended to match integer $i \in [1, N]$.

Second, the library $\mathcal{L}$ contains exactly the following rules.

- Rule $g_{\text{select}(k,\, j)} \lhd \langle p_{m^k_j - 1}$ **to** $p_{n^k_j} \rangle$ for every $k \in \{1, \ldots, m\}$ and every $j \in \{1, \ldots, |P_k|\}$ that amounts to requiring that if the j-th interval $[m^k_j, n^k_j]$ of pack P_k is selected, then it is covered;
- Rule $g_{\text{pack}(k)} \lhd \texttt{OR}(g_{\text{select}(k,\, 1)}, \ldots, g_{\text{select}(k,\, |P_k|)})$, for every $k \in \{1, \ldots, m\}$ requiring to select one of the $|P_k|$ intervals in the pack P_k;
- Rule $g_{\text{union}} \lhd \texttt{AND}(g_{\text{pack}(1)}, \ldots, g_{\text{pack}(m)})$ expressing that one must select an interval in each pack P_k;

Example 12. For the PIC instance of Example 11, we get trace

$$t = \{p_0\}\{p_1\}\{p_2\}\{p_3\}\{p_4\}\{p_5\}\{p_6\}\{p_7\}\{p_8\}\{p_9\}$$

and the following library:

$$\begin{cases} g_{\text{union}} & \lhd \texttt{AND}(g_{\text{pack}(1)}, g_{\text{pack}(2)}, g_{\text{pack}(3)}) \\ g_{\text{pack}(1)} & \lhd \texttt{OR}(g_{\text{select}(1,\,1)}, g_{\text{select}(1,\,2)}) \\ g_{\text{pack}(2)} & \lhd \texttt{OR}(g_{\text{select}(2,\,1)}, g_{\text{select}(2,\,2)}, g_{\text{select}(2,\,3)}) \\ g_{\text{pack}(3)} & \lhd \texttt{OR}(g_{\text{select}(3,\,1)}) \\ g_{\text{select}(1,\,1)} & \lhd \langle p_0 \textbf{ to } p_6 \rangle \\ g_{\text{select}(1,\,2)} & \lhd \langle p_4 \textbf{ to } p_9 \rangle \\ g_{\text{select}(2,\,1)} & \lhd \langle p_0 \textbf{ to } p_3 \rangle \\ g_{\text{select}(2,\,2)} & \lhd \langle p_3 \textbf{ to } p_6 \rangle \\ g_{\text{select}(2,\,3)} & \lhd \langle p_6 \textbf{ to } p_7 \rangle \\ g_{\text{select}(3,\,1)} & \lhd \langle p_3 \textbf{ to } p_4 \rangle \end{cases}$$

The obtained instance $\langle \mathcal{L}, t \rangle$ is computed in polynomial time from the PIC instance $\langle I, P_1, \ldots, P_m \rangle$. Clearly, the instance $\langle \mathcal{L}, t \rangle$ of the attack tree synthesis problem is positive if, and only if, the original PIC instance $\langle I, P_1, \ldots, P_m \rangle$ is positive. Indeed, there is a correspondence between the choice of intervals in packs, and the children of nodes labelled by $g_{\text{pack}(1)}, \ldots, g_{\text{pack}(m)}$ whose respective semantics exhibits m subtraces that cover the full trace t.

5.3 NP-membership of the synthesis problem

The following table shows the correspondence between some refinement rules and context-free grammars (CFG) rules in formal languages. Notice that there is no grammar rules counterpart for refinement rules with an `AND` operator.

Refinement rule	CFG production rule
$g \lhd \langle \iota \textbf{ to } \gamma \rangle$	$X \to a$
$g \lhd \texttt{OR}(g_1, g_2)$	$X \to Y \mid Z$
$g \lhd \texttt{SAND}(g_1, g_2)$	$X \to YZ$

Still, we are able to design an algorithm based on a variant of the classic bottom-up parsing algorithm "Cocke–Younger–Kasami algorithm" (CYK) [Kas66,You67,Sip97]. The original algorithm answers whether some input context-free grammar can generate some input word. It relies on a *dynamic programming* solution that computes, for each subword by increasing length, the set of non-terminals that generate it.

Algorithm design As in CYK, we handle sets $\texttt{Goals}[i, j]$ that collect goals of $\mathcal{G}$ that derives the subtrace $t[i, j]$. Nevertheless, we cannot rely on the mere dynamic programming anymore since the three operators do not necessarily make use of decreasing intervals. The following example illustrates the phenomenon with an artificial example of library.

Example 13. For $Prop = \{p_1, p_2, p_3, p_4\}$, take trace $t = \{p_1\}\{p_2\}\{p_3\}\{p_4\}$ and the following library $\mathcal{L}$:

$$\begin{cases} \rho_1 : g \lhd \texttt{OR}(g') & \rho_5 : g \lhd \langle p_2 \textbf{ to } p_3 \rangle \\ \rho_2 : g' \lhd \texttt{SAND}(g, g) & \rho_6 : g' \lhd \langle p_1 \textbf{ to } p_1 \rangle \\ \rho_3 : g \lhd \texttt{SAND}(g', g) & \rho_7 : g' \lhd \langle p_1 \textbf{ to } p_2 \rangle \\ \rho_4 : g \lhd \texttt{SAND}(g', g'') & \rho_8 : g'' \lhd \langle p_3 \textbf{ to } p_4 \rangle \end{cases}$$

Figure 6 shows an $\mathcal{L}$-attack tree for the trace t. Although the nodes marked * and the node marked ** are at different levels in the tree, we will see that both arise when computing $\texttt{Goals}[1, 3]$ to parse subtrace $t[1, 3] = \{p_1\}\{p_2\}\{p_3\}$.

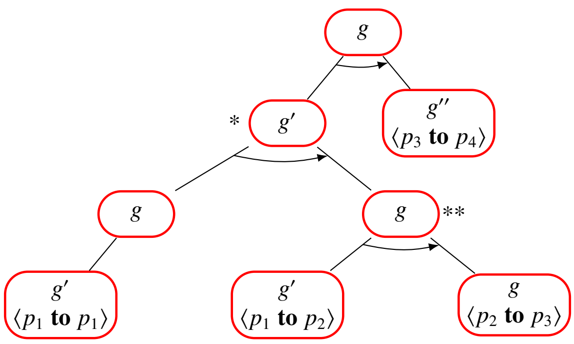

Fig. 6. An $\mathcal{L}$-attack tree for t.

Let us zoom on a bottom-up parsing of the trace t, by successively increasing the length of the subintervals $[i, j]$ to compute $\texttt{Goals}[i, j]$ that parses $t[i, j]$.

During the treatment of the 1-length interval $[1, 1]$, g' is put in Goals$[1, 1]$ thanks to Rule ρ_6, which allows next to add g by Rule ρ_1; Goals$[2, 2]$ and Goals$[3, 3]$ are empty.

We skip the computation for intervals of length 2, and focus on the treatment of interval $[1, 3]$: in order to obtain the subtree of Figure 6 rooted at node marked * for subtrace $t[1, 3]$, the parsing procedure should have added goal g in Goals$[1, 3]$ according to Rule ρ_3 (corresponding to the marked ** node) before adding g' (node *) thanks to Rule ρ_2. But because of the mutual recursivity of the rules, it seems difficult to know *a priori* which of Rule ρ_2 and Rule ρ_3 should be considered first.

In order to face the potential inability to exhibit a hierarchy of the rules for an arbitrary input library, we propose an algorithm that iterates on rules until stabilisation for each interval of the input trace.

Importantly, the ability to solve the synthesis problem even for libraries with mutual recursivity between rules is not a mere technical achievement but a true need: indeed, libraries may be fed incrementally by uncoordinated experts, which prevents us from requiring any sort of (in)dependencies between rules. Thus restricting to non recursive libraries (as for the museum Example 9) would be a very limited solution.

Regarding the technical aspects of our algorithm, the parsing of SAND-rules is handled with a minor adaptation of the CYK algorithm because of the tiny difference between classic concatenation and synchronized concatenation. On the contrary, since AND-rules of libraries do not have any counterpart in CF grammars, we resort to a novel method based on non-deterministically guessing one interval per subgoal, hence a non-deterministic algorithm.

Algorithm pseudo-code Algorithm 1 presents the pseudo-code of our non-deterministic algorithm that decides the attack tree synthesis in polynomial-time.

As in CYK, we consider each interval $[i, j]$ of $[1, n]$ by increasing length (line 1), and we compute Goals$[i, j]$ (in the **repeat-until** loop, lines 2-18) that is a set of goals that match $t[i, j]$ (possibly the set of exactly all such goals when the *right* non-deterministic choices are taken).

We iterate over all the rules of the library and update Goals$[i, j]$ according to the semantics given in Definition 4. For a rule $g \lhd \langle \iota \textbf{ to } \gamma \rangle$, we add the goal g to Goals$[i, j]$ when ι holds at time i and γ holds at time j. For a rule $g \lhd$ OR$(g_1, \dots, g_m)$, as long there is a goal g_k in Goals$[i, j]$, we add g to Goals$[i, j]$. For a rule $g \lhd$ SAND(g_1, g_2), if there is a midposition t between time i and j such that g_1 is in Goals$[i, t]$ and g_2 is in Goals$[t, j]$, we add g to Goals$[i, j]$. For a rule $g \lhd$ AND$(g_1, \dots, g_m)$, we first non-deterministically choose intervals $I_1, \dots, I_m$ included in $[i, j]$. In the case $I_1, \dots, I_m$ is a covering of $[i, j]$ and goals $g_1, \dots, g_m$ are respectively in Goals$[I_1], \dots,$ Goals$[I_m]$, we add g in Goals$[i, j]$. Note that if g is added in Goals$[i, j]$ then the rule $g \lhd$ AND$(g_1, \dots, g_m)$ can be applied to construct an attack tree. The reverse is false: it might be the case that the rule $g \lhd$ AND$(g_1, \dots, g_m)$ can be applied although g is not added to Goals$[i, j]$. Nevertheless, if the rule $g \lhd$ AND$(g_1, \dots, g_m)$ can be applied then there is an execution in which the goal g is added to Goals$[i, j]$.

At the end, the input is accepted exactly when Goals$[1, n]$ is not empty, that is, when the algorithm found that there is an attack tree for the full trace t.

Proposition 1 states the main properties of Algorithm 1.

Algorithm 1 `attackTreeSynthesis`($\mathcal{L}, t$): it has an accepting execution iff there is an $\mathcal{L}$-attack tree whose semantics contains t.

```
1:  for all intervals [i, j] of [1, n] by increasing length do
2:    repeat
3:      for all rules ρ in ℒ do
4:        match ρ do
5:          case g ◁ ⟨ι to γ⟩:
6:          if t(i) ⊨ ι and t(j) ⊨ γ then
7:            Goals[i, j] := Goals[i, j] ∪ {g}
8:          case g ◁ OR(g_1, ..., g_m):
9:            if there is 1 ≤ k ≤ m and g_k ∈ Goals[i, j] then
10:             Goals[i, j] := Goals[i, j] ∪ {g}
11:         case g ◁ SAND(g_1, g_2):
12:           if there is i≤t≤j such that g_1∈Goals[i, t] and g_2∈Goals[t, j] then
13:             Goals[i, j] := Goals[i, j] ∪ {g}
14:         case g ◁ AND(g_1, ..., g_m):
15:           non-deterministically choose  I_1, ..., I_m ⊆ [i, j]
16:           if I_1, ..., I_m covers [i, j] and g_1 ∈ Goals[I_1] and ... and g_m ∈ Goals[I_m] then
17:             Goals[i, j] := Goals[i, j] ∪ {g}
18:   until Goals[i, j] stabilises
19: if (Goals[1, n] ≠ ∅) accept else reject
```

Proposition 1.

1. *Executions of* `attackTreeSynthesis`($\mathcal{L}, t$) *have length in poly*(*size*($\mathcal{L}$) + $|t|$).
2. *(Soundness) If there is an accepting execution of* `attackTreeSynthesis`($\mathcal{L}, t$), *then there is an* $\mathcal{L}$*-attack tree* τ *such that* $t \in L(\tau)$.
3. *(Completeness) If there is an* $\mathcal{L}$*-attack tree* τ *such that* $t \in L(\tau)$*, then there is an accepting execution of* `attackTreeSynthesis`($\mathcal{L}, t$).

Proof. Since at each iteration of the **repeat-until** loop (lines 2-18) the set `Goals`$[i, j]$ is increasing and bounded by finite $\mathcal{G}$ (and the rest is clearly polynomial), we obtain Point 1. Also, the invariant "for all executions of `attackTreeSynthesis`($\mathcal{L}, t$), `Goals`$[i, j]$ is included in the set of goals that derive $t[i, j]$" entails Point 2. Finally, it suffices to consider the execution that chooses the right intervals at line 15 to get Point 3.

By Proposition 1, the attack tree synthesis problem is in NP.To achieve the proof of Theorem 1, it remains to restrict to libraries with bounded-arity `AND`-rules.

5.4 Libraries with bounded-arity `AND`-rules

It can be observed that the combinatorics of the unbounded `AND` operator contributes to the problem's complexity. By bounding the `AND` operator arity in library $\mathcal{L}$, the resulting subclass of the synthesis problem falls into P.

To see this, observe that bounding the `AND` operator arity yields a polynomial number of covers, so that line 15 of Algorithm 1 can be replaced by a **for**-loop over all covers that executes a polynomial number of times in the arity m.

6 Conclusion

We have presented a mathematical setting that addresses an attack tree synthesis problem. In this contribution, we rely on a formal trace semantics of attack trees inspired from the path semantics proposed, *e.g.,* in [APK17,APK18]. Our setting exploits the ontology of library whose rules describe how a subgoal can be refined into a combination of subgoals; such combinations rely on any of the classic tree operators `OR`, `SAND`, and `AND`. The synthesis problem has two inputs: a library and a trace. It consists in building an attack tree whose refinements are provided by the input library and whose semantics contains the input trace. We have established that the (associated decision) problem is NP-complete. However, the proposed algorithm is only polynomial in the size of the trace. This is good news for the two following reasons. First, traces might be long objects (*e.g.,* log files). Second, the exponential blow up caused by the arity of `AND` rules in libraries should be tamed in practice: the library is often fixed, and a manually entered `AND`-rule in this library is unlikely to have a huge arity.

Regarding synthesis, our algorithm can be easily extended to keep track of subtrees: each time a goal is added in `Goals`, there is a matching subtree that we could build – as done for the classic CYK algorithm to return the syntactic tree of a parsed word. This is very classic in dynamic programming and can still be exploited in our case.

Recently, new operators have been proposed to combine subgoals, among which weak variants of existing operators, as done in [MP19] and [PFWT19]. It can be shown that our algorithm extends without increasing the complexity for latter these operators.

This contribution opens several perspectives both theoretical and practical.

Theoretical level (1) We can investigate the use of first-order formulas in atomic goals $\langle \iota \textbf{ to } \gamma \rangle$, which would encompass the kinds of rules in [JLMRC18] and [IPHK15]. We foresee the need for pattern-matching techniques or Robinson's unification that may impact the theoretical complexity of the problem. (2) We may also relax the problem by not synthesizing a single tree, but a minimal number of trees where each one parses a piece of the input trace – this can be formalized). (3) We have to go beyond the case of a single trace, and synthesize a tree whose semantics contains (or equals) an input finite set of traces. This has already been addressed in [GJM+17] for the restricted case of `OR` and `SAND`-rules only, and regrettably with an incomplete procedure; the authors write that their procedure "either generates a correct tree or aborts" (in contrary, our approach is complete, see Point 3 of Proposition 1).

Practical level We foresee two main tracks. The first track regards the lengthy traces arising from concrete log files. Even if our algorithm is polynomial in this parameter, scalability is still an issue. We may explore abstractions of traces (*e.g.,* modulo stuttering equivalence), or subclasses of libraries with efficient parsing methods (*e.g.,* of the type LL(1)). The second track is ambitious and aims at bridging the gap between formal libraries and libraries in practice, such as the knowledge base of adversary tactics MITTRE ATT&CK[2]. We are not aware of any significant advance but of a humble recent degree project [ÅS19][3]. This topic should become very hot in the near future.

[2] https://attack.mitre.org/.

[3] http://www.diva-portal.org/smash/get/diva2:1350884/FULLTEXT01.pdf

References

APK17. Maxime Audinot, Sophie Pinchinat, and Barbara Kordy. Is My Attack Tree Correct? In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 83–102. Springer, 2017.

APK18. Maxime Audinot, Sophie Pinchinat, and Barbara Kordy. Guided design of attack trees: A system-based approach. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 61–75. IEEE Computer Society, 2018.

APSW18. Maxime Audinot, Sophie Pinchinat, François Schwarzentruber, and Florence Wacheux. Deciding the non-emptiness of attack trees. In *Graphical Models for Security - 5th International Workshop on Graphical Models for Security, Oxford, UK - July 8, 2018*, pages 25–38, 2018.

ÅS19. Oscar Åberg and Edvin Sparf. Validating the meta attack language using mitre att&ck matrix, 2019.

Aud18. Maxime Audinot. *Assisted design and analysis of attack trees*. PhD thesis, Université de Rennes 1, 2018.

BB09. EAC Advisory Board and Standards Board. Election Operations Assessment – Threat Trees and Matrices and Threat Instance Risk Analyzer (TIRA). 2009.

BK08. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

BKMS12. Alessandra Bagnato, Barbara Kordy, Per Håkon Meland, and Patrick Schweitzer. Attribute decoration of attack-defense trees. *Int. J. Secur. Softw. Eng.*, 3(2):1–35, April 2012.

FFG+16. Marlon Fraile, Margaret Ford, Olga Gadyatskaya, Rajesh Kumar, Mariëlle Stoelinga, and Rolando Trujillo-Rasua. Using attack-defense trees to analyze threats and countermeasures in an atm: A case study. In Jennifer Horkoff, Manfred A. Jeusfeld, and Anne Persson, editors, *The Practice of Enterprise Modeling*, pages 326–334, Cham, 2016. Springer International Publishing.

GJM+17. Olga Gadyatskaya, Ravi Jhawar, Sjouke Mauw, Rolando Trujillo-Rasua, and Tim A. C. Willemse. Refinement-Aware Generation of Attack Trees. In *STM*, volume 10547 of *LNCS*, pages 164–179. Springer, 2017.

HKT13. J. B. Hong, D. S. Kim, and T. Takaoka. Scalable attack representation model using logic reduction techniques. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 404–411, July 2013.

HMU07. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

IPHK15. Marieta Georgieva Ivanova, Christian W. Probst, René Rydhof Hansen, and Florian Kammüller. Attack Tree Generation by Policy Invalidation. In *WISTP*, volume 9311 of *LNCS*, pages 249–259. Springer, 2015.

JLMRC18. Ravi Jhawar, Karim Lounis, Sjouke Mauw, and Yunior Ramírez-Cruz. Semi-automatically augmenting attack trees using an annotated attack tree library. In *International Workshop on Security and Trust Management*, pages 85–101. Springer, 2018.

Kas66. Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*, 1966.

MP19. Heiko Mantel and Christian W. Probst. On the meaning and purpose of attack trees. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 184–199. IEEE, 2019.

(NE15. National Electric Sector Cybersecurity Organization Resource (NESCOR). Analysis of Selected Electric Sector High Risk Failure Scenarios, Version 2.0. 2015.

PAV14. Sophie Pinchinat, Mathieu Acher, and Didier Vojtisek. Towards Synthesis of Attack Trees for Supporting Computer-Aided Risk Analysis. In *SEFM Workshops*, volume 8938 of *LNCS*, pages 363–375. Springer, 2014.

PAV15. Sophie Pinchinat, Mathieu Acher, and Didier Vojtisek. ATSyRa: An Integrated Environment for Synthesizing Attack Trees – (Tool Paper). In *GraMSec*, volume 9390 of *LNCS*, pages 97–101. Springer, 2015.

PFWT19. Sophie Pinchinat, Barbara Fila, Florence Wacheux, and Yann Thierry-Mieg. Attack trees: A notion of missing attacks. In *Graphical Models for Security - 6th International Workshop, GraMSec@CSF 2019, Hoboken, NJ, USA, June 24, 2019, Revised Papers*, pages 23–49, 2019.

Sch99. Bruce Schneier. Attack Trees: Modeling Security Threats. *Dr. Dobb's Journal of Software Tools*, 24(12):21–29, 1999.

SCPS19. Abdallah Saffidine, Sébastien Lê Cong, Sophie Pinchinat, and François Schwarzentruber. The Packed Interval Covering Problem Is NP-complete. *CoRR*, abs/1906.03676, 2019.

Sip97. Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

Tov84. Craig A. Tovey. A simplified np-complete satisfiability problem. *Discrete Applied Mathematics*, 8(1):85 – 89, 1984.

VNN14. Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson. Automated generation of attack trees. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 337–350, 2014.

WAFP19. Wojciech Wideł, Maxime Audinot, Barbara Fila, and Sophie Pinchinat. Beyond 2014: Formal methods for attack tree-based security modeling. *ACM Computing Surveys*, 52(4), 2019. (accepted, to be published).

You67. Daniel H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189 – 208, 1967.

A Intractability of Pack Interval Covering

This section is devoted to proving strong NP-completeness of PIC, even under some restrictions. To do so, we will use a reduction from 2p1n≤3-sat, the restriction of sat where each variable has exactly two positive occurrences and one negative occurrence, and where each clause contains no more than 3 literals. Recall that 2p1n≤3-sat is NP-complete [Tov84, Theorem 2.1]. The problem 2p1n≤3-sat is defined by:

- Input: A finite collection of clauses $\{C_j\}_{1\leq j\leq m}$ over variables $\{p_i\}_{0\leq i<n}$, where each variable p_i appears twice as a positive literal and once as a negative literal, and each clause contains up to 3 literals.
- Output: Is the collection of clauses satisfiable?

As a running example, consider the following.

Example 14. The collection of clauses $\{(p_0 \vee p_1 \vee p_2), (p_0 \vee \neg p_1 \vee p_3), (\neg p_0 \vee p_2), (\neg p_3 \vee \neg p_2), (p_1 \vee p_3)\}$ is a positive instance of 2p1n≤3-sat. It can be satisfied by letting p_0, p_1, p_2 be assigned *true*, and p_3 be assigned *false*.

We are now equipped to prove our main result.

Theorem 3. *The Pack Interval Covering problem (*PIC*) is strongly* NP*-complete, even with all of the following restrictions, all packs have size 2, all intervals have length no more than 2, and all elements occur in no more than 3 intervals.*

Proof. Establishing membership in NP is straightforward. The following polynomial time non-deterministic algorithm solves PIC: guess an interval in each pack, then verify in polynomial time that the union of these intervals covers $[1, N]$.

For hardness, assume given $\{C_j\}_{1\leq j\leq m}$ an instance of 2p1n≤3-sat, over variables $\{p_i\}_{0\leq i<n}$. We build an equisatisfiable instance of PIC in polynomial time as follows.

Let $N = 2n + m$ so that the target interval is $[1, 2n + m]$ and define 3 packs for each variable as follows. For each variable p_i, let c^i_1 and c^i_2 (resp. c^i_3) be the indices of the two clauses (resp. unique clause) where variable p_i occurs as a positive literal (resp. negative literal). The packs corresponding to p_i are

$$
\begin{aligned}
P^i_1 &= \{\{2i + 1\}; \{2n + c^i_1\}\},\\
P^i_2 &= \{\{2i + 2\}; \{2n + c^i_2\}\},\\
Q^i &= \{[2i + 1, 2i + 2]; \{2n + c^i_3\}\}.
\end{aligned}
$$

The intuition behind the construction is that for $0 \leq i < n$, covering $[2i + 1, 2i + 2]$ ensures that variable p_i is assigned no more than one value, while for $1 \leq j \leq m$, covering $2n + j$ ensure that clause C_j is satisfied. Therefore, covering $[1, N]$ ensures that a variable assignment satisfying all clauses can be found.

The reduction is indeed polynomial since the size of obtained instance of PIC is linear in the size of the 2p1n≤3-sat instance. The magnitude of the interval bounds is polynomial in $m + n$. As such, encoding the integers in unary does not blow up the size of the resulting instance more than a polynomial factor. Thus the reduction shall

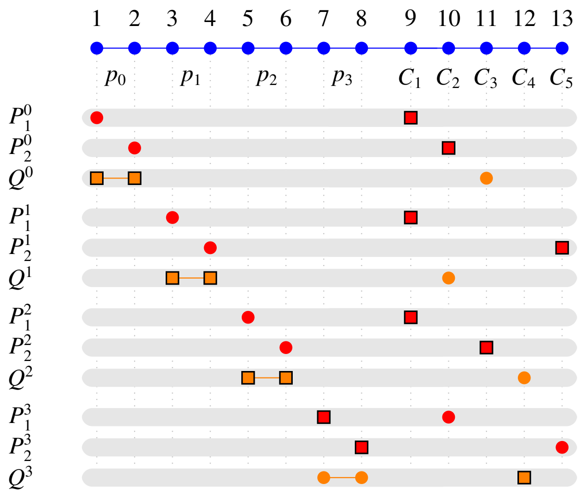

Fig. 7. The PIC instance associated to the 2P1N≤3-SAT instance in Example 14, ie., $\{(p_0 \vee p_1 \vee p_2), (p_0 \vee \neg p_1 \vee p_3), (\neg p_0 \vee p_2), (\neg p_3 \vee \neg p_2), (p_1 \vee p_3)\}$. A selection corresponding to p_0, p_1, p_2 being assigned *true* and p_3 being assigned *false* is indicated with square symbols instead of disks. One can verify that each column contains at least one square, so this selection indeed covers the target interval $[1, 13]$.

establish strong NP-hardness of PIC. Furthermore, all packs have size 2, all intervals have length no more than 2, and since each clause contains no more than 3 literals, all elements occur in no more than 3 intervals.

Example 15 (Example 14 continued). For the collection of clauses of Example 14, we get $N = 2 \times 4 + 5 = 13$ and the obtained packs can be seen in Figure 7. Each pack corresponds to a literal occurence in the 2P1N≤3-SAT instance (the rows), the interval $[1, 8]$ is made of pairs corresponding to the variables p_0 to p_3 (the first 8 columns), and each integer in the interval $[9, 13]$ corresponds to a clause (the last 5 columns). Just like the 2P1N≤3-SAT instance it is based on, this PIC instance is positve. It can be covered by selecting the intervals indicated with squares in Figure 7. Each pack contains exactly one selected (squared) interval, and each column is covered by at least one selected (squared) interval.

It remains to establish that this reduction is correct: that it maps positive 2P1N≤3-SAT instances to positive PIC instances and that it maps negative 2P1N≤3-SAT instances to negative PIC instances.

First, assume the 2P1N≤3-SAT instance is positive and let v be a valuation that satisfies all clauses of this instance of 2P1N≤3-SAT. Let us build a selection of intervals covering $[1, N]$ in the following way. For any variable p_i, its value in v determines which intervals we select in packs P^i_1, P^i_2, and Q^i. If $v(p_i) = true$, we select $\{2n + c^i_1\}$, $\{2n + c^i_2\}$, and $[2i + 1, 2i + 2]$, otherwise, $v(p_i) = false$ and we select $\{2i + 1\}$, $\{2i + 2\}$, and $\{2n + c^i_3\}$, respectively. We prove that this selection covers every integer in $[1, 2n + m]$ by examining separately integers in $[1, 2n]$ and integers in $[2n + 1, 2n + m]$.

- Any element $y \in [1, 2n]$ can be written as $y = 2i+g$ with $i \in [0, n-1]$ and $g \in [1, 2]$. If $\nu(p_i) = \mathit{true}$ then y is covered by the selection in pack Q^i else $\nu(p_i) = \mathit{false}$ and y is covered by the selection in pack P^i_g.
- Any element $y \in [2n + 1, 2n + m]$ can be written as $y = 2n + j$ with $j \in [1, m]$. Since ν satisfies clause C_j, at least of one its literals holds. In other words, there exists a variable p_i such that either $c^i_1 = j$ and $\nu(p_i) = \mathit{true}$, $c^i_2 = j$ and $\nu(p_i) = \mathit{true}$, or $c^i_3 = j$ and $\nu(p_i) = \mathit{false}$. In the former two cases, y is covered by the selection in pack P^i_1 or in pack P^i_2 while in the latter case it is covered by the selection in pack Q^i.

Second, assuming that the obtained PIC instance is positive, we show that the original 2P1N≤3-SAT instance is positive as well. From a solution to the PIC instance, we derive a valuation ν by letting $\nu(p_i) = \mathit{true}$ iff $[2i + 1, 2i + 2]$ is selected in pack Q^i. To show that ν satisfies any clause C_j, let us identify a literal that holds in it. Since we have a solution to the PIC instance, the integer $2n + j$ is covered by the selection. Therefore, either $\{2n + j\}$ is selected in P^i_g for some $0 \leq i < n$ and $g \in [1, 2]$ and literal p_i appears in C_j, or $\{2n + j\}$ is selected in Q^i for some $0 \leq i < n$ and literal $\neg p_i$ appears in C_j. In the former case, consider the integer $2i + g$, it cannot be covered with the selection from pack P^i_g since interval $\{2n + j\}$ was chosen instead. The only other way to cover $2i + g$ is to select $[2i + 1, 2i + 2]$ from the variable pack Q^i. Thus $\nu(p_i) = \mathit{true}$ and p_i satisfies clause C_j. In the latter case, $[2i + 1, 2i + 2]$ is not selected in pack Q^i, therefore $\nu(p_i) = \mathit{false}$ and $\neg p_i$ satisfies clause C_j. In both cases, ν is a model of C_j.