Refinement-Aware Generation of Attack Trees

Olga Gadyatskaya[1], Ravi Jhawar[1 2], Sjouke Mauw[1], Rolando Trujillo-Rasua[1], and Tim A. C. Willemse[3]

[1] SnT and University of Luxembourg,
Esch-sur-Alzette, Luxembourg
olga.gadyatskaya, sjouke.mauw, rolando.trujillo @uni.lu
[2] ILNAS
Esch-sur-Alzette, Luxembourg
ravi.jhawar@ilnas.etat.lu
[3] Eindhoven University of Technology,
Eindhoven, the Netherlands
t.a.c.willemse@tue.nl

Abstract. Attack trees allow a security analyst to obtain an overview of the potential vulnerabilities of a system. Due to their refinement structure, attack trees support the analyst in understanding the system vulnerabilities at various levels of abstraction. However, contrary to manually synthesized attack trees, automatically generated attack trees are often not refinement-aware, making subsequent human processing much harder. The generation of attack trees in which the refined nodes correspond to semantically relevant levels of abstraction is still an open question. In this paper, we formulate the attack-tree generation problem and propose a methodology to, given a system model, generate attack trees with meaningful levels of abstraction.

1 Introduction

Attack trees are a well-known graphical security model [Sch99]. They are widely used in industry and academia for handling threat modeling and security risk assessment [Sho14], as they help the analysts to structure the reasoning, facilitate communications across the board, and can store succinctly very complex threat scenarios [FFG$^+$16]. Yet, the process of creating an attack tree is quite lengthy, tedious, and error-prone [FFG$^+$16,Sho14]. It can be facilitated by applying industry threat catalogues [FFG$^+$16] and security knowledge bases [GLPS14], but these information sources may be unavailable for particular organizations or too generic to be useful. This is why recently researchers started to develop techniques for generating attack trees automatically [VNN14,IPHK15,HKT13,PAV15,Gad15].

Automatic generation of attack trees can be interpreted as model transformation. The initial model is typically a domain-specific language specifying the system components, their interactions, and the attacker's goal as an undesired state of the system, and formalizing attack paths towards the goal. In that regard, the attack-tree generation problem consists in encoding all attacks achieving a common goal into an attack tree model. However, this problem formulation overlooks one of the strengths of attack trees: its *refinement structure*.

Refining a goal into subgoals is an intuitive process for humans, used in attack trees and other visual languages, e.g., mind maps. That makes attack trees easily readable and comprehensible by a simple top-down inspection, as it allows the analyst to understand the attack potential at various levels of abstraction. This dimension, however, almost completely escapes in the literature on attack-tree generation, as we discuss in the following short overview of relevant approaches.

Vigo et al. [VNN14] generate trees from a process calculus system model by translating algebraic specifications into formulae and backward-chaining these formulae into a formula for the attacker's goal success. Reachability-based approaches, such as [IPHK15,Gad15,HKT13], transform system models into attack trees using information about connected elements in the model. In essence, these approaches reason that the attacker can reach the desired location from any system location adjacent to it. This reasoning is applied recursively to traverse complete attack paths. The main drawback of the techniques proposed in [VNN14,IPHK15,HKT13,Gad15] is that they do not leverage the refinement structure of attack trees, when parent nodes are more abstract than child nodes. In fact, [VNN14] does not provide any meaning to intermediate nodes, which only serve to express how child nodes are combined, while [IPHK15,Gad15,HKT13] have intermediate nodes at the same level of abstraction as child nodes, representing actions in the system model.

The attack traces-based approaches rely on a set of successful traces that capture transitions from the initial state to the state in the system in which the attacker has achieved the goal. The basic idea of generating successful attack paths has been explored in, e.g., [RA00,SHJ$^+$02], where the authors applied model-checking to network system models. Dawkins and Hale [DH04] have generated attack trees from network attack graphs (a formalism different from attack trees [SHJ$^+$02]) by finding minimum cut sets for successful attack paths (traces). This approach also does not offer a refinement structure, and each branch in a generated attack tree corresponds to a sequence of vulnerability exploitations.

The ATSyRA approach [PAV14,PAV15] synthesizes attack trees from attack graphs. It requires that the analyst first defines a set of actions at several abstraction levels in the system model, and a set of rules for refinement of higher-level actions into combinations of lower-level ones. This action hierarchy allows to transform successful attack paths in the attack graph (generated by model-checking) into an attack tree, containing precise actions as leaf nodes, while intermediate nodes represent more abstract actions. This tree enjoys a refinement structure that is more familiar to the human analyst, but the analyst still has to define the refinement relation herself; it is not created automatically.

We conclude that there exists a gap between manual and automatic generation of attack trees that has not been covered in literature yet. Automatic generation approaches work with concrete attacks, while manual creation of attack trees focuses on the refinement of goals into subgoals. In this paper, we address this gap by formalising the attack-tree generation problem that connects both properties, namely that of encoding a set of attacks and that of respecting a given refinement structure.

Summary of contributions. This paper presents the following main results:

– We formally define the attack-tree generation problem as a task to generate a tree with an expected meaning that respects a given refinement structure.
– We propose an approach for generating attack trees from traces of successful attacks in a system model. Our approach utilizes a heuristic for encoding and decomposing attack traces that is based on the edge biclique problem [Pee03]. Furthermore, we derive the refinement structure from an abstraction relation on system predicates.
– We demonstrate the feasibility of our approach with a running example of a network security scenario.

2 The attack-tree generation problem

In this section we first formally introduce attack trees and the notion of a refinement specification. Next, we define what it means for an attack tree to satisfy a refinement specification and we formulate the *attack-tree generation problem*. Informally, this problem requires the derivation of a tree with a given semantics, that satisfies a given refinement specification.

Intuitively, an attack tree defines how higher (parent) nodes are interpreted through lower (child) nodes. The interpretations are defined by the refinement operators: `OR` specifies that if any of the child nodes is achieved, then the immediate parent node is also achieved; and `AND` defines that all child nodes need to be achieved to achieve the parent node's goal [MO05]. We will consider also the sequential `AND` operator, or `SAND`, that demands that the goals of the child nodes are to be achieved in a particular order for achieving the parent node [JKM+15].

Formally, let $\mathbb{B}$ denote a set of *actions*, `OR` and `AND` be two unranked associative and commutative operators, and `SAND` be an unranked associative but non-commutative operator. A `SAND` attack tree t is an expression over the signature $\mathbb{B} \cup \{\mathtt{OR}, \mathtt{AND}, \mathtt{SAND}\}$, generated by the following grammar (for $b \in \mathbb{B}$):

$$t ::= b \mid b \lhd \mathtt{OR}(t, \dots, t) \mid b \lhd \mathtt{AND}(t, \dots, t) \mid b \lhd \mathtt{SAND}(t, \dots, t).$$

We use $\mathbb{T}_{\mathtt{SAND}}$ to denote all `SAND` attack trees generated by the grammar above. Different to the definition of `SAND` trees given in [JKM+15], we require every node in the tree to be annotated with an action. An action in a node typically provides a generic (sometimes vague) description of the type of attack, e.g. *get a user's credentials* or *impersonate a security guard*, which is helpful to a top-down interpretation of the tree. An expression like $b \lhd \mathtt{SAND}(t_1, \dots, t_n)$ denotes an attack tree of which the top node is labelled with action b, and which has n children $t_1, \dots, t_n$ that have to be executed sequentially.

Example 1. Figure 1 illustrates a simple `SAND` attack tree in which the goal of the attacker is to gain unauthorized access to a server. To achieve this goal, the

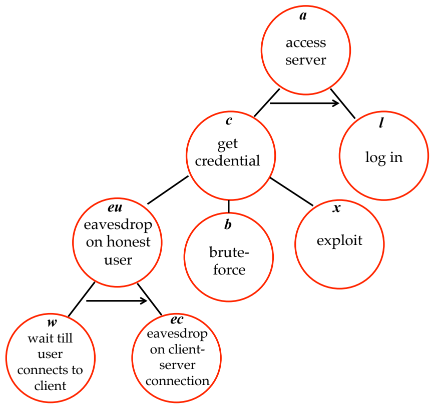

Fig. 1. A human-designed attack tree representing possible threat scenarios

attacker must first get a suitable credential for the server, and then, use this credential to log in remotely. A suitable credential can be obtained by eavesdropping on communications of an honest user, who knows the server password. Alternatively, the attacker can bruteforce the password on the server, or use an exploit to create a new password. Using shorthands for the action names, this tree can be represented by the following expression: $a \lhd \mathtt{SAND}(c \lhd \mathtt{OR}(eu \lhd \mathtt{SAND}(w, ec), b, x), l)$.

We define the auxiliary function *top* to obtain the action at the root node as follows (for $\varDelta \in \{\mathtt{OR}, \mathtt{AND}, \mathtt{SAND}\}$):

$$top(b) = top(b \lhd \varDelta(t, \ldots, t)) = b.$$

We say that t' is a subtree of t, denoted $t' \in t$, if $t = t'$ or $t = \varDelta(b, t_1, \ldots, t_n)$ and $t' \in t_i$ for some $i \in \{1, \ldots, n\}$, where $\varDelta \in \{\mathtt{OR}, \mathtt{AND}, \mathtt{SAND}\}$.

Given a semantical domain D, an attack-tree semantics S defines a function $[\cdot]_S \colon \mathbb{T}_{\mathtt{SAND}} \to D$. We denote semantic equivalence of two trees $t, t' \in \mathbb{T}_{\mathtt{SAND}}$ by $t =_S t'$, which means $[t]_S = [t']_S$. In this article we use the SP semantics [JKM+15] as the semantic domain for `SAND` attack trees. Notice, nevertheless, that our attack-tree generation problem formulation abstracts away from any concrete interpretation of the attack tree semantics.

The SP semantics encodes an attack tree as a set of *Series-Parallel graphs* (SP graphs). An SP graph is an edge-labeled directed graph with a *source* vertex and a *sink* vertex. The simplest SP graph has the form $u \xrightarrow{b} v$, where b is an edge label, u is the *source* vertex because it has no incoming edges, and v is the *sink* vertex because it has no outgoing edges. Any other SP graph is obtained as the composition of single-edge SP graphs.

Two composition operators are used to build SP graphs: the sequential composition operator $(\cdot)$ and the parallel composition operator $(\|)$. A sequential composition joins the sink vertex of a graph with the source vertex of the other graph. For example, given $G = u \xrightarrow{b} v$ and $G' = x \xrightarrow{z} y$, we obtain that

$G \cdot G' = u \xrightarrow{b} v \xrightarrow{z} y$. Note that the source vertex of G' has been replaced in $G \cdot G'$ by the sink vertex v of G. A parallel composition, instead, joins the source and the sink vertices of both graphs. For example, given $G = u \xrightarrow{b} v$ and $G' = x \xrightarrow{z} y$, the parallel composition $G \parallel G'$ gives the following SP graph.

$$\begin{tikzcd} u \arrow[r, bend left, "b"] \arrow[r, bend right, "z"'] & v \end{tikzcd}$$

In the SP semantics, edge labels represent basic actions in $\mathbb{B}$, and vertex labels are ignored. Hence a graph of the type $u \xrightarrow{b} v \xrightarrow{z} y$ is read as $\xrightarrow{b}\xrightarrow{z}$. Moreover, both composition operators are extended to sets of SP graphs as follows: given sets of SP graphs $\mathcal{G}_1, \dots, \mathcal{G}_k$,

$$\mathcal{G}_1 \parallel \mathcal{G}_2 \parallel \dots \parallel \mathcal{G}_k = \{G_1 \parallel \dots \parallel G_k \mid (G_1, \dots, G_k) \in \mathcal{G}_1 \times \dots \times \mathcal{G}_k\}$$
$$\mathcal{G}_1 \cdot \mathcal{G}_2 \cdot \dots \cdot \mathcal{G}_k = \{G_1 \cdot \dots \cdot G_k \mid (G_1, \dots, G_k) \in \mathcal{G}_1 \times \dots \times \mathcal{G}_k\}.$$

Definition 1. *Let $\mathbb{G}_{\mathcal{SP}}$ denote the set of SP graphs labeled with the elements of $\mathbb{B}$. The* SP semantics *for* `SAND` *attack trees is given by the function $[\![\cdot]\!]_{\mathcal{SP}} : \mathbb{T}_{\texttt{SAND}} \to \mathcal{P}(\mathbb{G}_{\mathcal{SP}})$, which is defined recursively as follows: for $b \in \mathbb{B}$, $t_i \in \mathbb{T}_{\texttt{SAND}}$, $1 \leqslant i \leqslant k$,*

$$[\![b]\!]_{\mathcal{SP}} = \{\xrightarrow{b}\}$$
$$[\![\texttt{OR}(t_1, \dots, t_k)]\!]_{\mathcal{SP}} = \bigcup_{i=1}^{k} [\![t_i]\!]_{\mathcal{SP}}$$
$$[\![\texttt{AND}(t_1, \dots, t_k)]\!]_{\mathcal{SP}} = [\![t_1]\!]_{\mathcal{SP}} \parallel \dots \parallel [\![t_k]\!]_{\mathcal{SP}}$$
$$[\![\texttt{SAND}(t_1, \dots, t_k)]\!]_{\mathcal{SP}} = [\![t_1]\!]_{\mathcal{SP}} \cdot \dots \cdot [\![t_k]\!]_{\mathcal{SP}}.$$

We kindly refer the reader to [JKM+15] for more details on the SP semantics.

Example 2. The `SAND` attack tree in Figure 1 has the following SP semantics: $\{\xrightarrow{w}\xrightarrow{ec}\xrightarrow{l}, \xrightarrow{b}\xrightarrow{l}, \xrightarrow{x}\xrightarrow{l}\}$. Note that the labels of the internal nodes are not represented in the SP semantics. Further note that the SP graphs occurring in this example are linear traces because the tree has no `AND` nodes.

Refinement specification. The transition from one level in an attack tree to the next level defines a *refinement*. More precisely, a refinement is an expression of the form $b \lhd \mathit{\Delta}(b_1, \dots, b_n)$, where $b, b_1, \dots, b_n \in \mathbb{B}$ and $\mathit{\Delta} \in \{\texttt{OR}, \texttt{AND}, \texttt{SAND}\}$. That is to say, a refinement corresponds to a tree of depth one. It follows that the set of refinements, denoted R, is a subset of the set of attack trees $\mathbb{T}_{\texttt{SAND}}$. In particular, the refinement of the root node of an attack tree is determined by the partial function $\mathit{ref} : \mathbb{T}_{\texttt{SAND}} \to R$, defined by

$$\mathit{ref}(b \lhd \mathit{\Delta}(t_1, \dots, t_n)) = b \lhd \mathit{\Delta}(\mathit{top}(t_1), \dots, \mathit{top}(t_n)).$$

This is a partial function, since the refinement of an attack tree that consists of a single node is not defined. This function can be generalized to non-root nodes, allowing us to determine the set of all refinements that occur in an attack tree. Therefore, we define the function $\mathit{refs}\colon \mathbb{T}_{\texttt{SAND}} \to \mathcal{P}(R)$, as follows:

$$\mathit{refs}(t) = \{\mathit{ref}(t') \mid t' \in t \wedge \neg\exists b \in \mathbb{B}\colon t' = b\}.$$

A *refinement specification* specifies which refinements should be satisfied by an attack tree. A refinement specification is simply defined as a set of refinements.Given an attack tree $t \in \mathbb{T}_{\texttt{SAND}}$ and a refinement specification $\rho \subseteq R$, we use $t \vdash \rho$ to denote that t *satisfies* ρ. We define satisfaction by $t \vdash \rho \iff \mathit{refs}(t) \subseteq \rho$. That is, a tree satisfies a refinement specification, if all refined actions in the tree also occur as refined actions in the specification.

Attack tree generation problem. Given an attack tree semantics and a refinement specification, the challenge is to design or derive an attack tree with this semantics that satisfies the refinement specification. We call this problem *the attack-tree generation problem.*

Definition 2 (The attack-tree generation problem). *Let S be an attack-tree semantics with semantic domain D. The* attack-tree generation problem *consists in, given a semantical element $d \in D$ and a refinement specification $\rho \subseteq R$, finding an attack tree $t \in \mathbb{T}_{\texttt{SAND}}$, such that $[t]_S = d$ and $t \vdash \rho$. Such a tree is called* correct with respect to a semantics and refinement specification (d, ρ).

Example 3. Given required semantics $\{\xrightarrow{w}\xrightarrow{ec}\xrightarrow{l}, \xrightarrow{b}\xrightarrow{l}, \xrightarrow{x}\xrightarrow{l}\}$ and refinement specification $\{a\lhd\texttt{SAND}(c,l), c\lhd\texttt{OR}(eu,b,x), eu\lhd\texttt{SAND}(w,ec), c\lhd\texttt{SAND}(p,q)\}$, a possible solution to the attack-tree generation problem is given in Figure 1. Note that the last refinement does not occur in the tree.

Clearly, an instance of the attack-tree generation problem may not have a solution. If it has a solution, the solution may not be unique. Depending on the purpose of the tree, the application domain, or even the taste of the designer, one could have a preference for a certain type of tree, aiming at, e.g., trees with minimal width, balanced trees or trees with a minimum number of leaf nodes.

The remainder of this paper is devoted to addressing the attack-tree generation problem.

3 Generating correct attack trees

In this section we will specialize the attack-tree generation problem by focusing only on `OR` and `SAND` nodes, and considering the semantic domain for attack trees to be the SP semantics [JKM+15]. Given this restriction, we develop an algorithm to generate correct attack trees using a greedy heuristic based on the *edge biclique* problem [Pee03].

The motivation for omitting the `AND` operator is the following. One of the inputs to the attack-tree generation problem is the intended semantics of the tree. We assume that the intended semantics is given by a set of traces, where each trace is an ordered sequence of actions. Such a set of traces could, e.g., be generated by a model checker that aims to reach the goal of the attacker [LMO15]. As traces are totally ordered, we can use the SAND operator to represent a trace and the OR operator to represent the choice between the alternative traces. Hence, starting from a set of traces, there is no need for the AND operator. An example of a trace model based on labelled transition systems is given in Sect. 4.

Properties of correct attack trees. Next we provide necessary and sufficient conditions for a tree to be correct. For the sake of simplicity, we focus on binary instances of the attack tree operators only. This simplifies the analysis and generalizes easily due to associativity of all operators.

Theorem 1. *Let $\mathcal{G}$ be a set of SP graphs with labels in $\mathbb{B}$, ρ a refinement specification, and t an attack tree of the form $b \lhd \mathtt{SAND}(t_l, t_r)$ (resp. $b \lhd \mathtt{OR}(t_l, t_r)$) where t_l and t_r are attack trees. The attack tree t is correct w.r.t. $(\mathcal{G}, \rho)$ if and only if there exist sets of SP graphs $\mathcal{G}_l$ and $\mathcal{G}_r$ such that all the following conditions are satisfied:*

1. *t_l is correct with respect to $(\mathcal{G}_l, \rho)$,*
2. *t_r is correct with respect to $(\mathcal{G}_r, \rho)$,*
3. *$\mathcal{G} = \mathcal{G}_l \cdot \mathcal{G}_r$ (resp. $\mathcal{G} = \mathcal{G}_l \cup \mathcal{G}_r$),*
4. *$b \lhd \mathtt{SAND}(top(t_l), top(t_r)) \in \rho$ (resp. $b \lhd \mathtt{OR}(top(t_l), top(t_r)) \in \rho$).*

Proof. ($\Rightarrow$) Let t be a correct tree w.r.t. $(\mathcal{G}, \rho)$ of the form $b \lhd \mathtt{SAND}(t_l, t_r)$ (resp. $\mathtt{OR}(t_l, t_r)$). Condition 4 holds by definition given that $t \vdash \rho$. Similarly we obtain that t_l and t_r must satisfy that $t_l \vdash \rho$ and $t_r \vdash \rho$, otherwise $t \nvdash \rho$. Condition 3 holds by definition of the SP semantics, where $[\![t]\!]_{\mathcal{SP}} = [\![t_l]\!]_{\mathcal{SP}} \cdot [\![t_r]\!]_{\mathcal{SP}}$ if t is of the form $b \lhd \mathtt{SAND}(t_l, t_r)$, $[\![t]\!]_{\mathcal{SP}} = [\![t_l]\!]_{\mathcal{SP}} \cup [\![t_r]\!]_{\mathcal{SP}}$ otherwise. Therefore, t_l and t_r are correct w.r.t. $([\![t_l]\!]_{\mathcal{SP}}, \rho)$ and $([\![t_r]\!]_{\mathcal{SP}}, \rho)$, respectively.

($\Leftarrow$) Now, let us assume that the four conditions above are satisfied. On the one hand, because t_l and t_r are correct w.r.t. $(\mathcal{G}_l, \rho)$ and $(\mathcal{G}_r, \rho)$, respectively, it follows that $\mathcal{G}_l = [\![t_l]\!]_{\mathcal{SP}}$ and $\mathcal{G}_r = [\![t_r]\!]_{\mathcal{SP}}$. Therefore, an attack tree t of the form $b \lhd \mathtt{SAND}(t_l, t_r)$ (resp. $b \lhd \mathtt{OR}(t_l, t_r)$) satisfies that $[\![t]\!]_{\mathcal{SP}} = \mathcal{G}_l \cdot \mathcal{G}_r = \mathcal{G}$ (resp. $[\![t]\!]_{\mathcal{SP}} = \mathcal{G}_l \cup \mathcal{G}_r = \mathcal{G}$). On the other hand, because $b \lhd \mathtt{SAND}(top(t_1), top(t_2)) \in \rho$ (resp. $b \lhd \mathtt{OR}(top(t_1), top(t_2)) \in \rho$) and t_1 and t_2 both satisfy ρ, we obtain that t satisfies ρ as well. This gives that t is correct w.r.t. $(\mathcal{G}, \rho)$. □

According to Theorem 1, a disjunctive refinement requires finding two subsets $\mathcal{G}_l$ and $\mathcal{G}_r$ that cover $\mathcal{G}$, i.e. $\mathcal{G}_l \cup \mathcal{G}_r = \mathcal{G}$. This is a fairly trivial task as, for example, a partition of a set is also a covering. However, a sequential conjunctive refinement requires finding a sequential decomposition of $\mathcal{G}$ in two sets $\mathcal{G}_l$ and $\mathcal{G}_r$ such that $\mathcal{G}_l \cdot \mathcal{G}_r = \mathcal{G}$. Clearly, such a decomposition is not always possible. Therefore, we focus on the problem of finding two sets $\mathcal{G}_l$ and $\mathcal{G}_r$ such that $\mathcal{G}_l \cdot \mathcal{G}_r \subseteq \mathcal{G}$ and $|\mathcal{G}_l \cdot \mathcal{G}_r|$ is maximum, which we call the *set decomposition problem.*

We tackle the set decomposition problem by reducing it to the *edge biclique problem* [Pee03], which benefits from well-known efficient algorithms [GG14] in the graph theory field. The edge biclique problem consists in finding, given a bipartite graph G, a biclique in G with maximum number of edges. A graph G is *bipartite* if its set of vertices can be partitioned into subsets V_1 and V_2 such that every edge in G connects a vertex in V_1 with a vertex in V_2. And G is said to be a *biclique* if every $(u, v) \in V_1 \times V_2$ is an edge in G. We usually write $G = (V_1 \cup V_2, E)$ to denote that G is bipartite with partite sets V_1 and V_2.

Theorem 2. *The set decomposition problem is polynomial-time reducible to the edge biclique problem, and vice-versa.*

Proof. ($\Rightarrow$) Let $\mathcal{G}$ be a non-empty set of SP graphs. Given an SP graph $\alpha = \xrightarrow{b_1} \dots \xrightarrow{b_n}$, let α_i^l and α_i^r denote the SP graphs $\xrightarrow{b_1} \dots \xrightarrow{b_i}$ and $\xrightarrow{b_{i+1}} \dots \xrightarrow{b_n}$, respectively. Let $G = (V, E)$ be a simple graph with set of vertices $V = \{\alpha_i^l | \alpha \in \mathcal{G} \wedge i < |\alpha|\} \cup \{\alpha_i^r | \alpha \in \mathcal{G} \wedge i < |\alpha|\}$ and set of edges $E = \{(\alpha_i^l, \beta_j^r) | \alpha_i^l \cdot \beta_j^r \in \mathcal{G}\}$. Now, let $G' = (U' \cup V', E')$ be a biclique in G. By construction of G we obtain the following two results. First, for every $(u, v) \in U' \times V'$ it holds that $u \cdot v \in \mathcal{G}$. Hence $U' \cdot V' \subseteq \mathcal{G}$. Second, for every pair of sets $\mathcal{G}_l$ and $\mathcal{G}_r$ such that $\mathcal{G}_l \cdot \mathcal{G}_r \subseteq \mathcal{G}$ it holds that $\mathcal{G}_l \subseteq U$ and $\mathcal{G}_r \subseteq V$. Hence the subgraph of G induced by the vertices $\mathcal{G}_l \cup \mathcal{G}_r$ is a biclique. Therefore, $G' = (U' \cup V', E')$ is a maximum biclique if and only if (U', V') is an optimal solution to the set decomposition problem.
($\Leftarrow$) Let $G = (U \cup V, E)$ be a bipartite graph and $\mathcal{G} = \{u \cdot v | u \in U \wedge v \in V \wedge (u, v) \in E\}$. Let $\mathcal{G}_l$ and $\mathcal{G}_r$ be a decomposition (not necessarily optimal) of $\mathcal{G}$, i.e. $\mathcal{G}_l \cup \mathcal{G}_r \subseteq \mathcal{G}$. As before, we obtain by construction the following two results. First, because $\mathcal{G}_l \subseteq U$ and $\mathcal{G}_r \subseteq V$, it follows that the subgraph in G induced by $\mathcal{G}_l \cup \mathcal{G}_r$ is a biclique. Second, for every biclique $G' = (U' \cup V', E')$ in G it holds that $U' \cdot V' \subseteq \mathcal{G}$. Therefore, $\mathcal{G}_l$ and $\mathcal{G}_r$ form an optimal decomposition of $\mathcal{G}$ if and only if the subgraph in G induced by $\mathcal{G}_l \cup \mathcal{G}_r$ is a maximum biclique. $\square$

From Theorem 2 we extract two conclusions. First, the set decomposition problem is NP-complete, given that the edge biclique problem is NP-complete [Pee03]. Second, we can use well-known approximation algorithms for the edge biclique problem to find approximate solutions for the set decomposition problem. Due its simplicity, in this article we use the greedy heuristic proposed by Gillis and Glineur [GG14]. A pseudocode description of such a heuristic is given in Figure 2.

```
procedure BICLIQUE($G = (X \cup Y, E)$)
    Let $Z$ be an empty set of vertices
    while $G$ is not bipartite do
        Let $u$ be any vertex in $G$ with maximum degree that is not contained in $Z$
        Let $W = X$ if $u \notin X$, $W = Y$ otherwise
        for all $v \in W$ such that $(u, v) \notin E$ do
            Remove $u$ from $G$ and the corresponding edges from $E$
        Remove isolated vertices from $G$
        Add $u$ to $Z$
    return $G$
```

Fig. 2. BICLIQUE is a greedy heuristic that approximates the edge biclique problem.

Example 4. To illustrate the procedure of decomposing a set of SP graphs via the reduction depicted in Theorem 2 and the Biclique heuristic, let us consider the following set $\mathcal{G} = \{\xrightarrow{a}\xrightarrow{a}, \xrightarrow{b}\xrightarrow{a}\xrightarrow{a}, \xrightarrow{b}\xrightarrow{a}\xrightarrow{c}, \xrightarrow{a}\xrightarrow{c}\}$. We first transform $\mathcal{G}$ into a graph G as indicated in Theorem 2. The resulting graph is depicted in Figure 3. Note that, for the sake of simplicity, we have omitted the arrow ($\rightarrow$) representing single-edge SP graphs in the vertex labels in G. By running the Biclique algorithm depicted in Figure 2, we obtain a subgraph of G that is a biclique. The obtained complete bipartite graph (see Figure 3) is then transformed into two sets of SP graphs by considering the vertex set partition. In the example, the two sets are $\mathcal{G}_l = \{\xrightarrow{a}, \xrightarrow{b}\xrightarrow{a}\}$ and $\mathcal{G}_r = \{\xrightarrow{a}, \xrightarrow{c}\}$. The pair of sets satisfies that $\mathcal{G}_l \cdot \mathcal{G}_r = \mathcal{G}$, because the biclique found by Biclique is optimal. Otherwise we can only guarantee that $\mathcal{G}_l \cdot \mathcal{G}_r \subsetneq \mathcal{G}$. A pseudocode description of the decomposition procedure explained in this example can be found in Figure 4.

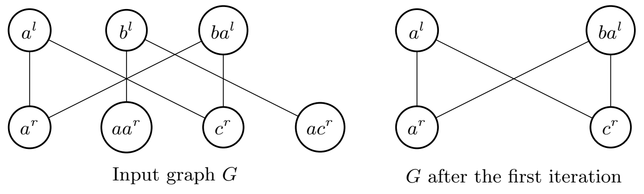

Fig. 3. An example of the execution of Biclique on graph G. The vertex with maximum degree chosen in this execution is a^l. The resulting graph is already a biclique.

```
procedure Decomposition($\mathcal{G}$)
    Let $G = (U \cup V, E)$ be an empty graph
    for all $\alpha \in \mathcal{G}$ do            ▷ Vertex set generation (see Theorem 2)
        for all $i = 1$ to $i = |\alpha| - 1$ do
            Add $\alpha_i^l$ to $U$ and $\alpha_i^r$ to $V$
    for all $(u, v) \in U \times V$ do            ▷ Edge set generation (see Theorem 2)
        if $u \cdot v \in \mathcal{G}$ then
            Add $(u, v)$ to $E$
    Let $G' = (U' \cup V', E')$ the output of Biclique on input $G$
    return $(U', V')$
```

Fig. 4. Biclique is a greedy heuristic that approximates the edge biclique problem.

Binary attack trees. We use the Decomposition procedure on a set of SP graphs to generate correct attack trees, with the peculiarity that the resulting tree is binary. The algorithm is given in Figure 5.

```
procedure GEN-BIN-TREE($(\mathcal{G}, \rho)$)
    if $\xrightarrow{b_1} \in \mathcal{G}$ for some $b_1 \in \mathbb{B}$ then          ▷ A single-edge SP graph exists in $\mathcal{G}$
        Let $t_2$ be the output of GEN-BIN-TREE($\mathcal{G} - \{\xrightarrow{b_1}\}, \rho$)
        Let $b \in \mathbb{B}$ such that $b \lhd \mathtt{OR}(b_1, top(t_2)) \in \rho$
        if $b$ exists then
            return $b \lhd \mathtt{OR}(\xrightarrow{b_1}, t_2)$
    else
        Let $(\mathcal{G}_l, \mathcal{G}_r)$ be the output of DECOMPOSITION on input $\mathcal{G}$
        if $\mathcal{G}_l \cdot \mathcal{G}_r = \mathcal{G}$ then
            Let $t_1$ be the output of GEN-BIN-TREE($\mathcal{G}_l, \rho$)
            Let $t_2$ be the output of GEN-BIN-TREE($\mathcal{G}_r, \rho$)
            Let $b \in \mathbb{B}$ such that $b \lhd \mathtt{SAND}(top(t_1), top(t_2)) \in \rho$
            if $b$ exists then
                return $b \lhd \mathtt{SAND}(t_1, t_2)$
        else
            Let $t_1$ be the output of GEN-BIN-TREE($\mathcal{G}_l \cdot \mathcal{G}_r, \rho$)
            Let $t_2$ be the output of GEN-BIN-TREE($\mathcal{G} - (\mathcal{G}_l \cdot \mathcal{G}_r), \rho$)
            Let $b \in \mathbb{B}$ such that $b \lhd \mathtt{OR}(top(t_1), top(t_2)) \in \rho$
            if $b$ exists then
                return $b \lhd \mathtt{OR}(t_1, t_2)$
    return fail
```

Fig. 5. GEN-BIN-TREE generates correct and binary attack trees.

The procedure GEN-BIN-TREE focuses on creating an attack tree t such that $[\![t]\!]_{\mathcal{SP}} = \mathcal{G}$, where $\mathcal{G}$ is a set of SP graphs given as input. Moreover, GEN-BIN-TREE guarantees that all refinements in the generated tree are in the refinement specification ρ, otherwise the algorithm aborts. Therefore, it follows that GEN-BIN-TREE either generates a correct tree or aborts.

It is worth remarking that GEN-BIN-TREE favours `SAND` refinements over `OR` refinements. The reason is that a `SAND` refinement requires solving the edge biclique problem. Thus, whenever a sequential decomposition of $\mathcal{G}$ is found, a `SAND` refinement is created.

Example 5. To illustrate the attack-tree generation approach, consider the `SAND` attack tree in Figure 1, whose SP semantics is $\mathcal{G} = \{\xrightarrow{w}\xrightarrow{ec}\xrightarrow{l}, \xrightarrow{b}\xrightarrow{l}, \xrightarrow{x}\xrightarrow{l}\}$. For the sake of simplicity, let us also consider the existence of a special action $\epsilon \in \mathbb{B}$ and a refinement specification ρ defined as the minimum set satisfying that $\epsilon \lhd \mathtt{OR}(b_1, b_2) \in \rho$ and $\epsilon \lhd \mathtt{AND}(b_1, b_2) \in \rho$ for every $b_1, b_2 \in \mathbb{B}$. This is for the moment an oversimplification of the role of the refinement specification. We defer the task of providing a tree with meaningful refinements to the next section.

By using the BICLIQUE procedure we obtain that $\mathcal{G}$ can be decomposed by $\mathcal{G}_l = \{\xrightarrow{w}\xrightarrow{ec}, \xrightarrow{b}, \xrightarrow{x}\}$ and $\mathcal{G}_r = \{\xrightarrow{l}\}$. The application of the GEN-BIN-TREE algorithm on input $\mathcal{G}_l$ gives the tree displayed in Figure 6. The same figure depicts the tree obtained on input $\mathcal{G}_r$. The sequential composition of the two trees is finally the output of GEN-BIN-TREE on input $\mathcal{G}$.

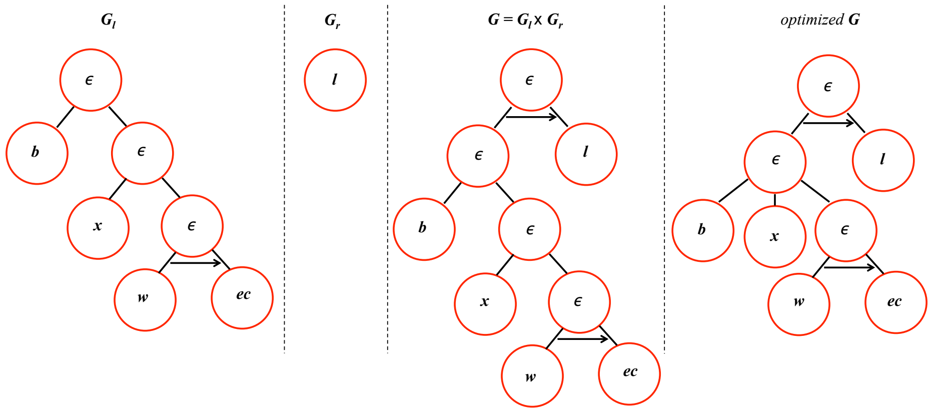

Fig. 6. The attack-tree generation process for the SP-semantics given in Example 2.

We observe that Algorithm Gen-Bin-Tree generates trees that, although correct, use a rather artificial binary branching structure. We thus use semantics-preserving transformation rules to optimize the structure of the tree. A semantics-preserving transformation rule is a total function $r\colon \mathbb{T}_{\mathtt{SAND}} \to \mathbb{T}_{\mathtt{SAND}}$ such that $\forall t \in \mathbb{T}_{\mathtt{SAND}}\colon [\![t]\!]_{\mathcal{SP}} = [\![r(t)]\!]_{\mathcal{SP}}$. In our approach we use the following rule: for every $\Delta \in \{\mathtt{OR}, \mathtt{SAND}\}$ and every $t = b \lhd \Delta(t_1, \dots, t_k)$,

$$r(t) = \begin{cases} b \lhd \Delta(t_1^1, \dots, t_{k(1)}^1, \dots, t_1^k, \dots, t_{k(k)}^k) & \text{If } t_i = b_i \lhd \Delta(t_1^i, \dots, t_{k(i)}^i), \forall i \in \{1, \dots, k\} \\ t & \text{otherwise.} \end{cases}$$

This simply amounts to aggregating nodes whenever allowed by associativity of the operator. Figure 6 shows the result of the application of this rule to the binary tree obtained by Gen-Bin-Tree algorithm. Note that the semantics-preserving transformation rule does not take into account the refinement specification ρ. Thus, if ρ is an arbitrary set of refinements and it is not closed under the SP-semantics equivalence relation, the optimized tree may not be correct, while being semantically-equivalent to the original tree.

4 Specifying a system and refinement relation

The attack-tree generation problem is based on two inputs: an intended semantics and a refinement specification. In this section we show how both can be obtained from an LTS-based system model. Finally, we illustrate our methodology through a simple example.

System specification. Labelled transition systems are used to describe the behaviour of a system by defining the transitions that bring a system from one state into another. Formally, a *Labelled Transition System* (LTS) is a quadruple $(\mathcal{S}, \Sigma, \rightarrow, s_0)$, where $\mathcal{S}$ is a set of *states*; Σ is a set of *labels*; $\rightarrow\colon \mathcal{S} \times \Sigma \times \mathcal{S}$ is a *transition relation*; $s_0 \in \mathcal{S}$ is the *initial state*.

We define a state as a set of predicates. A predicate defines a mutable property of the system, such as $knows(Alice, psw)$, which means that *Alice* knows password *psw*. States are denoted by $[p_1, \dots, p_n]$, where $p_1, \dots, p_n$ are the predicates that determine the state. If s is a state, then by $s[p_1, \dots, p_n]$ we mean the state s augmented with predicates $p_1, \dots, p_n$. If a predicate is preceded by a $\neg$ symbol it means that the predicate is removed from the state. For instance, if $s_0 = [p_1, p_2]$, then $s_0[p_3, \neg p_1] = [p_2, p_3]$.

The states will be used to label the nodes of the attack tree that will be generated, so we will equate the set of states and the set of actions in the attack tree, $\mathbb{B} = \mathcal{S}$.

The transition relation is defined through *transition rules.* Figure 7 shows some example transition rules. Every transition rule contains a condition (above the horizontal line) and a conclusion (below the line). The name of a transition rule is given left of the line. The condition consists of a number of predicates that must be present in the current state to enable the transition rule. The conclusion describes the state change when the transition occurs. The old state is described left of the transition arrow and the new state right of the arrow. The arrow is labeled with the event that describes the transition. The predicates may contain variables, which are implicitly universally quantified.

Refinement specification. The second input to our algorithms is the refinement relation. We first define a partial order $\sqsubseteq$ on $\mathbb{B}$, which we call an *abstraction relation.* Given that states are sets of predicates, we can define this abstraction relation as set inclusion, $s \sqsubseteq s' \iff s \subseteq s'$. If $s \sqsubseteq s'$, we say that s is *more abstract* than s'. From this abstraction relation we can derive a refinement specification, as follows.

Definition 3 (Abstraction-based refinement specification). *Let $\mathbb{B}$ be a set of actions with abstraction relation $\sqsubseteq$. The* abstraction-based refinement relation *is the smallest refinement relation $\rho_\sqsubseteq$ that satisfies (for $\forall b, b_1, \dots, b_n \in \mathbb{B}$):*

$$\text{if } b \sqsubseteq b_1 \wedge \dots \wedge b \sqsubseteq b_n \text{ then } b \lhd \mathtt{OR}(b_1 \cdots b_n) \in \rho_\sqsubseteq, \text{ and}$$
$$\text{if } b \sqsubseteq b_n \text{ then } b \lhd \mathtt{SAND}(b_1 \cdots b_n) \in \rho_\sqsubseteq.$$

This definition expresses that the attacker's goal of an `OR` node must be more abstract than the attacker's goals of its children, and that the attacker's goal of a `SAND` node must be more abstract than the goal of its right-most child.

Note that for more elaborated definitions of the system state, the abstraction relation can be modified accordingly. We could, for instance, consider a state consisting of two sets of predicates describing desired and undesired properties.

Network security example. We consider a set of machines M on a simple network and a set of human actors A that can use these machines. We also consider a set of credential records R, and a set of user terminals $T \subseteq M$.

Further, we consider the following set of predicates:

- *located*: $A \times M$ determines to which machines actors are connected;

$$[\mathit{startTerm}]\frac{}{s \xrightarrow{\mathit{startTerm}(a,t)} s[\mathit{located}(a,t)]}$$

$$[\mathit{loggingInRem}]\frac{\mathit{located}(a,m), \mathit{connected}(m,m_1), \mathit{stores}(m_1,r), \mathit{knows}(a,r)}{s \xrightarrow{\mathit{loggingInRem}(a,m,m_1,r)} s[\mathit{located}(a,m_1)]}$$

$$[\mathit{loggingOut}]\frac{\mathit{located}(a,m),}{s \xrightarrow{\mathit{loggingOut}(a,m)} s[\neg \mathit{located}(a,m)]}$$

$$[\mathit{exploiting}]\frac{\mathit{located}(a,m), \mathit{connected}(m,m_1),}{s \xrightarrow{\mathit{exploiting}(a,m,m_1,r)} s[\mathit{stores}(m_1,r), \mathit{knows}(a,r)]}$$

$$[\mathit{bruteforcingPsw}]\frac{\mathit{located}(a,m), \mathit{connected}(m,m_1)\mathit{stores}(m_1,r)}{s \xrightarrow{\mathit{bruteforcingPsw}(a,m,m_1,r)} s[\mathit{knows}(a,r)]}$$

$$[\mathit{eavesdropping}]\frac{\mathit{located}(a,m), \mathit{located}(a_1,m), \mathit{connected}(m,m_1), \mathit{knows}(a,r), \mathit{stores}(m_1,r)}{s \xrightarrow{\mathit{eavesdropping}(a,a_1,m,m_1,r)} s[\mathit{knows}(a_1,r)]}$$

Fig. 7. Transition rules for the example ($a, a_1 \in A$, $m, m_1 \in M$, $t \in T$ and $r \in R$).

- *connected*: $M \times M$ defines directly connected machines;
- *stores*: $M \times R$ identifies credentials accepted by a machine.
- *knows*: $A \times R$ determines which credentials are known to actors.

Figure 7 presents a set of transition rules for this system. The first three rules define the behaviour of legitimate users, and the other three rules introduce actions for attackers.

As an example system we consider the set M consisting of just two machines, client C and server S, and the set of terminals T to contain only C. We consider two actors *Alice* and *Mallory*, and two credentials *psw* and *psw₁*.
Initial state:
$s_0 = [\mathit{located}(\mathit{Mallory}, C), \mathit{connected}(C,S), \mathit{stores}(S, \mathit{psw}), \mathit{knows}(\mathit{Alice}, \mathit{psw})]$.
Final state: Any state s_f that contains $\mathit{located}(\mathit{Mallory}, S)$.
Traces: We consider the following three traces that lead to a successful attack.

Trace T^1:
$s_0 \xrightarrow{\mathit{exploiting}(\mathit{Mallory},C,S,\mathit{psw_1})} s_0[\mathit{stores}(S, \mathit{psw_1}), \mathit{knows}(\mathit{Mallory}, \mathit{psw_1})]$
$\xrightarrow{\mathit{loggingInRem}(\mathit{Mallory},C,S,\mathit{psw_1})} s_0[\mathit{stores}(S, \mathit{psw_1}), \mathit{knows}(\mathit{Mallory}, \mathit{psw_1}), \mathit{located}(\mathit{Mallory}, S)]$

Trace T^2:
$s_0 \xrightarrow{\mathit{bruteforcingPsw}(\mathit{Mallory},C,S,\mathit{psw})} s_0[\mathit{knows}(\mathit{Mallory}, \mathit{psw})]$
$\xrightarrow{\mathit{loggingInRem}(\mathit{Mallory},C,S,\mathit{psw})} s_0[\mathit{knows}(\mathit{Mallory}, \mathit{psw}), \mathit{located}(\mathit{Mallory}, S)]$

Trace T^3:
$s_0 \xrightarrow{\mathit{startTerm}(\mathit{Alice},C)} s_0[\mathit{located}(\mathit{Alice}, C)]$

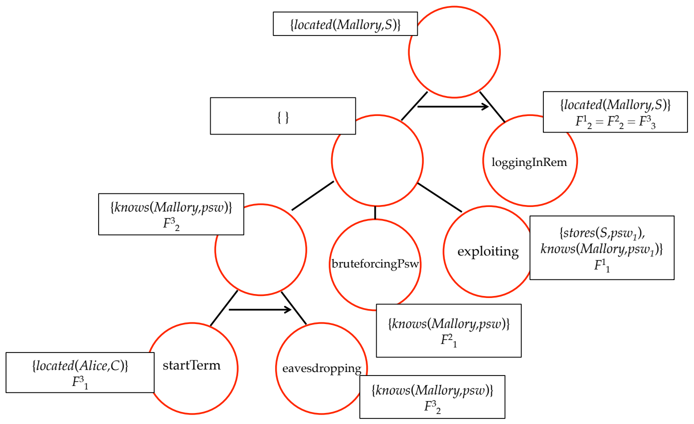

Fig. 8. Generated attack tree for the network example.

$$\xrightarrow{eavesdropping(Alice,Mallory,C,S,psw)} s_0[located(Alice, C), knows(Mallory, psw)]$$
$$\xrightarrow{loggingInRem(Mallory,C,S,psw)} s_0[located(Alice, C), knows(Mallory, psw), located(Mallory, S)]$$

Intuitively, the tree for *Mallory* accessing the server S would be as presented in Fig. 1: *Mallory* can attempt to eavesdrop on *Alice* to learn *psw* or to bruteforce *psw*; or he can exploit S to create a new credential psw_1. Next we show the tree obtained by using our approach for automated attack-tree generation.

- The path in T^1 is characterised by $b_1^1 b_2^1$, where $b_1^1 = (\varnothing, \{stores(S, psw_1), knows(Mallory, psw_1)\}$ and $b_2^1 = (\{\varnothing, \{located(Mallory, S)\}$.
- The path in T^2 is characterised by $b_1^2 b_2^2$, where $b_1^2 = (\varnothing, \{knows(Mallory, psw)\}$, $b_2^2 = (\varnothing, \{located(Mallory, S)\})$.
- The path in T^3 is characterised by $b_1^3 b_2^3 b_3^3$, where $b_1^3 = (\varnothing, \{located(Alice, C)\})$, $b_2^3 = (\varnothing, \{knows(Mallory, psw)\})$, and $b_3^3 = (\varnothing, \{located(Mallory, S)\})$.

Based on these runs, our approach generates the tree presented in Fig. 8. In this figure, the node labels identified by our approach are in boxes. Furthermore, note that we have also labelled leaf nodes in a more meaningful way (labels in the red circles) by using the corresponding actions (labels in the LTS) of the system transitions. At the same time, most of the labels for the intermediate nodes in the generated tree are also informative, as they specify only the facts relevant for achieving the attack's success in a particular subtree.

Note that one intermediate node has a label that represents an empty set of facts, as there are no common facts for its children. This node has to be interpreted by the analyst as a combination of its children nodes. Yet, our approach can be extended to be able to suggest meaningful labels also for such nodes. This can be realized, e.g., through supporting first-order logic facts with quantifiers, such as $\{\exists r \in R\colon knows(Mallory, r), stores(S, r)\}$.

It is worth remarking that the generated tree is identical in structure to the human-designed tree (Fig. 1). However, this is not guaranteed for other scenarios.

5 Conclusions

In this paper we have introduced the attack-tree generation problem as a task of constructing a correct attack tree that both has some expected meaning and respects a pre-defined refinement relation. This problem definition supports a more uniform treatment of the issues arising in both manual creation of attack trees and automatic generation from system models. Furthermore, we have developed a solution for this problem that utilizes an abstraction-based refinement specification derived from a system model and a set of traces representing successful attack scenarios in the model to generate a correct attack tree.

The trees we generate are refinement-aware, and thus provide more insight to the analyst than attack trees generated by previously proposed approaches, such as [IPHK15,Gad15,HKT13,VNN14]. Furthermore, our approach derives the refinement relation from the system model itself, and so it reduces the load on the analyst in comparison to the ATSyRA approach [PAV15].

The novelty of our approach consists also in the labelling technique for intermediate and leaf nodes. Our labelling is based on the facts about the system state that the attacker wants to achieve or avoid in order to realize the attack. Our running example of the network security case has shown that the proposed generation and labelling technique is practical and yields meaningful attack trees.

To continue this work, we plan to integrate a model checker for obtaining system traces, and to implement the generation algorithm in the open-source attack tree software ADTool [GJK+16].

Acknowledgements. The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement number 318003 (TREsPASS) and from the Fonds National de la Recherche Luxembourg under grant C13/IS/5809105 (ADT2P).

References

[DH04] J. Dawkins and J. Hale. A systematic approach to multi-stage network attack analysis. In *Proc. of Inf. Assurance Workshop*. IEEE, 2004.

[FFG+16] M. Ford, M. Fraile, O. Gadyatskaya, R. Kumar, M. Stoellinga, and R. Trujillo-Rasua. Using attack-defense trees to analyze threats and countermeasures in an ATM: A case study. In *Proc. of PoEM*. Springer, 2016.

[Gad15] O. Gadyatskaya. How to generate security cameras: Towards defence generation for socio-technical systems. In *Proc. of GraMSec*. Springer, 2015.

[GG14] N. Gillis and Fr. Glineur. A continuous characterization of the maximum-edge biclique problem. *J. Global Optimization*, 58(3):439–464, 2014.

[GJK+16] O. Gadyatskaya, R. Jhawar, P. Kordy, K. Lounis, S. Mauw, and R. Trujillo-Rasua. Attack trees for practical security assessment: Ranking of attack scenarios with ADTool 2.0. In *Proc. of QEST*. Springer, 2016.

[GLPS14] H. Ghani, J. Luna, I. Petkov, and N. Suri. User-centric security assessment of software configurations: A case study. In *Proc. of ESSoS*. Springer, 2014.

[HKT13] J. B. Hong, D. S. Kim, and T. Takaoka. Scalable attack representation model using logic reduction techniques. In *Proc. of TrustCom*. IEEE, 2013.

[IPHK15] M. G. Ivanova, C. W. Probst, R. R. Hansen, and F. Kammuller. Transforming graphical system models to graphical attack models. In *Proc. of GraMSec*. Springer, 2015.

[JKM$^+$15] R. Jhawar, B. Kordy, S. Mauw, S. Radomirović, and R. Trujillo-Rasua. Attack trees with sequential conjunction. In *Proc. of SEC*, volume 455 of *IFIP AICT*. Springer, 2015.

[KMRS14] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer. Attack-defense trees. *Oxford Univ. Press J. Logic and Computation*, 24(1):55–87, 2014.

[LMO15] G. Lenzini, S. Mauw, and S. Ouchani. Security analysis of socio-technical physical systems. *Elsevier Computers & Electrical Engineering*, 2015.

[MO05] S. Mauw and M. Oostdijk. Foundations of Attack Trees. In *Proc. of ICISC*, pages 186–198. Springer, 2005.

[PAV14] S. Pinchinat, M. Acher, and D. Vojtisek. Towards synthesis of attack trees for supporting computer-aided risk analysis. In *Proc. of SEFM and Workshops*, volume 8938 of *LNCS*, pages 363–375, 2014.

[PAV15] S. Pinchinat, M. Acher, and D. Vojtisek. ATSyRa: an integrated environment for synthesizing attack trees. In *Proc. of GraMSec*. Springer, 2015.

[Pee03] R. Peeters. The maximum edge biclique problem is NP-complete. *Discrete Appl. Math.*, 131(3):651–654, September 2003.

[RA00] R. W Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proc. of S&P Symposium*, pages 156–165. IEEE, 2000.

[RKT12] A. Roy, D. S. Kim, and K. Trivedi. Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees. *Security and Comm. Networks*, 5(8), 2012.

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

[SHJ$^+$02] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proc. of S&P Symposium*, pages 273–284. IEEE, 2002.

[Sho14] A. Shostack. *Threat modeling: Designing for security.* Wiley, 2014.

[VNN14] R. Vigo, F. Nielsen, and H. R. Nielson. Automated generation of attack trees. In *Proc. of CSF*, pages 337–350. IEEE, 2014.