Beyond 2014: Formal methods for attack tree-based security modeling

WOJCIECH WIDEŁ, Univ Rennes, INSA Rennes, CNRS, IRISA, Rennes, France
MAXIME AUDINOT, Univ Rennes, CNRS, IRISA, Rennes, France
BARBARA FILA, Univ Rennes, INSA Rennes, CNRS, IRISA, Rennes, France
SOPHIE PINCHINAT, Univ Rennes, CNRS, IRISA, Rennes, France

Attack trees are a well-established and commonly used framework for security modeling. They provide a readable and structured representation of possible attacks against a system to protect. Their hierarchical structure reveals common features of the attacks and enables quantitative evaluation of security, thus highlighting the most severe vulnerabilities to focus on while implementing countermeasures. Since in real-life studies attack trees have a large number of nodes, their manual creation is a tedious and error-prone process, and their analysis is a computationally challenging task. During the last half decade, the attack tree community witnessed a growing interest in employing formal methods to deal with the aforementioned difficulties. We survey recent advances in graphical security modeling, with focus on the application of formal methods to the interpretation, (semi-)automated creation, and quantitative analysis of attack trees and their extensions. We provide a unified description of existing frameworks, compare their features, and outline interesting open questions.

CCS Concepts: • **Security and privacy** → **Formal security models**; *Logic and verification*;

Additional Key Words and Phrases: Attack trees, attack–defense trees, graphical security modeling, formal methods, quantitative analysis of security, automatic generation of security models, model checking, logics

ACM Reference format:
Wojciech Wideł, Maxime Audinot, Barbara Fila, and Sophie Pinchinat. 2019. Beyond 2014: Formal methods for attack tree-based security modeling. *ACM Comput. Surv.* 1, 1, Article 1 (January 2019), 35 pages.
https://doi.org/10.1145/3331524

1 INTRODUCTION

Whether it is for representing vulnerabilities of various voting schemes [35], analyzing security of critical infrastructures in the electric sector [82], classifying ATM-related frauds [37], or quantifying cost, difficulty, and time of attacks against an RFID-based goods management system [18], attack trees and their derivatives have been successfully adopted by the industrial sector as a means of modeling and evaluation of security. This graphical formalism, inspired by threat logic trees [107] and introduced by Schneier [101], owes its popularity to its simplicity, on the one hand, and a large range of potential applications, on the other hand. The simple but powerful idea behind attack trees is to recursively decompose an often complex attack scenario into sub-scenarios that can be described and quantified more easily. With attack trees, one may capture, in a single model,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
0360-0300/2019/1-ART1 $15.00
https://doi.org/10.1145/3331524

scenarios involving technical, physical, as well as human vulnerabilities, and thus perform an in-depth analysis of security. Finally, in [101] Schneier also drafted an efficient, bottom-up procedure (certainly inspired by techniques developed for fault trees [45]) exploiting the tree-like nature of attack trees and allowing for their quantitative and qualitative evaluation.

Over the last twenty years, the original model of Schneier has been further improved, from the scientific and practical perspective. Numerous formal semantics for attack trees have been proposed [54, 57, 68, 81]; more powerful algorithms for quantitative analysis allowing for multiparameter evaluation and taking dependencies between attack steps into account have been developed, cf., Section 5, 6, and 7; the expressive power of attack trees has been augmented with countermeasures [25, 51, 68, 99], attacker profiles [39, 80], and a description of the analyzed system [15, 16, 56]. Furthermore, a number of commercial and academic prototype tools have been implemented to support real-life security analysis with attack tree-based models [52].

From the theoretical perspective, the most recent trend in the domain is to enhance the design and analysis of attack trees with formal methods. In this paper, the term *formal methods* is broadly understood, and covers standard verification techniques, e.g., model checking, automata theory, logics, SAT, SMT or constraint solving, graph theory, as well as computational and optimization algorithms based on integer linear programming, Bayesian networks, genetic algorithms, etc. The objective of this survey is to *give an overview of existing approaches integrating attack tree-based modeling and formal methods* in the following three dimensions

(1) **semantical approaches**, where the objective is to give a rigorous, mathematical *meaning* to the (extended) attack tree model;
(2) **generation approaches**, aiming at a (semi-)automated *creation* of attack(–defense) trees;
(3) **quantitative approaches**, focusing on algorithms and techniques for *quantitative analysis* of security.

We mainly consider two models: attack trees and their extensions with dependent nodes and attack–defense trees augmenting attack trees with the nodes representing countermeasures. For each presented approach, we first state its purpose and indicate which formal method has been used in combination with attack or attack–defense modeling. An illustrative diagram is given to visualize the approach in a schematic way. Then, we provide a short overview of the employed formal method and explain how it has been incorporated into the graphical security modeling and analysis. We also discuss whether there are any particular assumptions under which the framework can be applied. Finally, we compare the presented approaches with the other ones dealing with similar objectives.

While selecting the articles included in this overview paper, our aim was to be complementary with respect to existing surveys in the domain of graphical security (and safety) modeling. Therefore, we cover only the approaches using formal methods and being introduced between 2014 and 2018. An exhaustive state of the art on DAG-based security modeling until 2013 has been presented in [69]. The reader interested in usability aspects, practical applications, and computer tools for graphical security modeling is referred to [52]. Finally, the approaches focusing on formal methods for fault trees and their usage in safety modeling, have been reviewed in [100] and [65].

The intended public of this work is two-fold. First, we target the scientific community interested in graphical security modeling. In this case, the paper's aim is to gather relevant existing approaches, present them in a unified way, and compare their features. Our objective is also to follow up the work initiated in [69] and [52] and keep an up-to-date description of the research performed in the field. Second, engineers and developers of tools supporting graphical security modeling and analysis will find in this paper an overview of how formal methods can be exploited to enhance the security evaluation based on graphical models. Here, our goal is to provide a comprehensible

summary of approaches that might be pertinent for their work and give pointers to the sources where more details can be found. Especially for this second group of readers, we made an effort to keep the frameworks' descriptions self-contained, in the sense that individual sections can be read independently. Finally, essential aspects of the described frameworks are gathered in three comparative tables: Table 1 for semantical approaches, Table 3 for generation approaches, and Table 4 for quantitative approaches. Their purpose is to quickly guide the reader towards the solution that suits best their needs.

This survey is structured as follows. We start by recalling the preliminary information on attack trees and attack–defense trees, in Section 2. The objective is to briefly present the two models and set up the vocabulary used in the rest of the paper. Section 3 is devoted to frameworks addressing the meaning and the expressiveness of the security models (semantical approaches), and Section 4 gathers approaches aiming at their (semi-)automated creation (generation approaches). The next three sections are dedicated to the quantitative analysis of security (quantitative approaches). In Section 5, we present methods enhancing the original bottom-up procedure to deal with multi-objective optimization problems and to capture trees with repeated nodes. Section 6 focuses on frameworks using timed automata and related model checking techniques and tools. Finally, Section 7 concentrates on probabilistic analysis of security, involving stochastic games, probabilistic model checking, and Bayesian networks. We summarize the scientific visibility of the graphical security modeling area in Section 8, where we also outline interesting, open research directions that are awaiting to be explored in the near future.

2 ATTACK(–DEFENSE) TREES IN A NUTSHELL

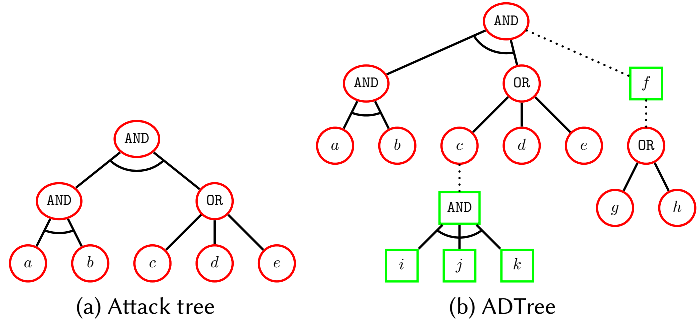
Attack trees [101] are a model for hierarchical representation of attack scenarios. Formally, they are *rooted trees* with *labeled nodes*. The labels of the nodes represent *goals* of the attacker, with the label of the root node corresponding to the main goal of the modeled scenario. If achieving a particular goal requires from the attacker to achieve some other sub-goals, then the node labeled with that goal is called *refined*. The basic model of attack trees admits two types of refinements: OR and AND. To achieve the goal of an AND node (a conjuctively refined node), one needs to achieve sub-goals of all of its children, whereas the goal of an OR node (a disjunctively refined node) is achieved when the sub-goal of at least one of its children is achieved. Another often considered refinement is the sequential refinement (SAND). Similarly as in the case of the conjunctive refinement, achieving the goal of an SAND node requires achieving sub-goals of all of its children, but in the given order. If a node is not refined, which in the case of attack trees is equivalent with it being a leaf node, then the goal it represents is called a *basic action*.

Attack–defense trees (ADTrees) [68] enhance the expressive power of attack trees by explicitly depicting goals of another actor – a defender – in the model. In a scenario represented by an ADTree, a goal of an actor can be *countered* by a goal of the other actor. According to the terminology introduced in [68], the root actor is called the *proponent* and the other actor is the *opponent*. The proponent's aim is to achieve the root goal, while the opponent tries to make it impossible.

Examples of symbolic attack tree and ADTree are given in Fig. 1. According to the standard convention, red circles depict nodes of the attacker and green rectangles of the defender. Edges connecting an AND node with its children are joined with an arc. Dotted edges connect countermeasures with the nodes whose goals they counter. In order for the attacker to achieve the main goal of the scenario modeled with the attack tree from Fig. 1a, they need to execute both basic actions a and b, and at least one of the actions c, d, or e. Such combinations of actions that are sufficient to achieve the root goal of an attack tree are often called *attack vectors*, *attack strategies*, or simply *attacks*. For instance, a, b, c and a, b, d are two examples of attack vectors covered by the attack

tree from Fig. 1a. These vectors, however, are not sufficient for the attacker to achieve the root goal of the ADTree from Fig. 1b. The vector a, b, c will not work, because the execution of c might be countered by the defender performing actions i, j, and k. To make use of the vector a, b, d, the attacker needs to additionally execute action g or h, to counterattack the countermeasure f.

Even though the models of attack trees and ADTrees appear to be very intuitive and self-explanatory, their size and formal treatment are often problematic. Thus, it seems natural to take advantage of the power of formal methods to support the interpretation, creation, and analysis of attack tree-based models. The remaining sections of this paper present the results of the scientific effort that has been undertaken in this direction.



(a) Attack tree (b) ADTree

Fig. 1. An attack tree and an attack–defense tree

3 FORMAL INTERPRETATIONS OF ATTACK TREES

The aim of attack trees is to represent complex security scenarios in an intuitive and easy to understand way. The power of the model relies on two factors: the *labels* of the nodes that may express any type of digital, physical, or human-related security concerns, and an intuitive notion of *decomposition* of complex goals into simpler sub-goals and basic actions. Anyone who read the original article of Schneier [101] will be able to draw a meaningful attack tree. However, to formally analyze such trees, one needs to express them in a rigorous way, i.e., give them a *formal interpretation*. Assigning mathematical objects to attack trees helps addressing a wide range of problems, including enumerating all attack vectors covered by the tree, checking whether two structurally different trees represent the same security scenario, comparing whether one tree contains more information than another one, identifying paths in the analyzed system that correspond to potential attack vectors, and verifying the quality of the attack tree refinements.

Classically, the following two interpretations are used for attack trees: the *propositional semantics*, like in [60], where an attack tree is interpreted as a Boolean function representing the structure of the tree, and the *multiset semantics* [81], interpreting an attack tree as a set of multisets modeling the attack vectors covered by the tree. Unfortunately, none of these interpretations is rich enough to capture the sequential behavior of the attacker, i.e., to model the SAND refinement.

This section provides an overview of more expressive, mathematical interpretations of attack trees, that have been recently developed with the goal of differentiating between AND and SAND refinements. We start with the *series-parallel* (SP) *graph* semantics (Section 3.1) that conservatively extends the multiset semantics to attack trees with SAND. Section 3.2 investigates how *linear logic* can be used to interpret and compare attack trees. Both the SP and the linear logic-based semantics are independent of (the model of) the analyzed system. They also abstract away from the meaning of the labels of the attack tree nodes. In contrast, the work presented in Section 3.3 takes the model of the system to be analyzed into account. It proposes a semantics that relies on *paths in the transition system* representing the analyzed real-life system, and expresses the labels of the attack

tree nodes in a formal way. This enables a verification of the correctness of an attack tree with respect to the analyzed system.

A comparative table of the mathematical interpretations of attack trees with SAND is given in Table 1. The approaches presented in this section are mainly concerned with the semantics providing a formal meaning to attack trees. Semantics that have been proposed to address specific quantitative problems, e.g., timed or probabilistic analysis of attacks, are described in Section 6 and 7.

Table 1. Mathematical interpretations of attack trees with SAND

Sec.	**Mathematical object**	AND **vs** SAND	**Tool**
3.1	Series-parallel graphs	Incomparable	SPTool [67]
3.2	Linear logic formulæ	AND may specialize SAND SAND may specialize AND	–
3.3	Paths in a transition system	SAND specializes AND	ATSyRA Studio [3]

3.1 Series-parallel interpretation: first formal foundations of attack trees with SAND

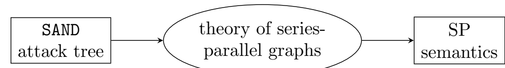


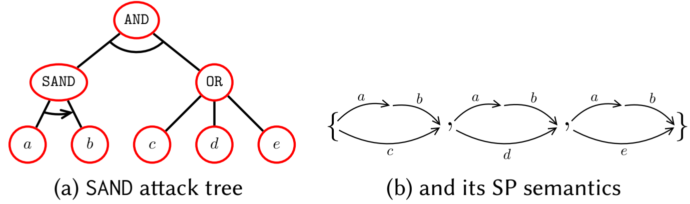
Fig. 2. SP semantics of [57]

The problem of ordering actions that compose an attack vector has been apparent for attack trees since their introduction in 1999. Even though original attack trees use only OR and AND refinements, already the very first examples of attack trees implicitly assume that actions under (some) AND nodes are ordered. For instance, consider the tree illustrated in Figure 8 of [101] representing attacks against a general computer system. In order to get a message that has been stored on the user's hard drive, the attacker needs to get access to the hard drive *and* read the file. Obviously, the action of accessing is a prerequisite for reading the file, so it needs to be performed first. In general, the problem of ordering actions in attack trees has been addressed in two ways: either AND is implicitly interpreted as an ordered operator (as in the example above) or an extra sequential operator, that we call SAND and depict with an arrow, is added to capture that some actions must be executed in a specific order, as proposed by Jhawar et al. in [57]. The objective of [57] is to provide mathematical foundations of attack trees extended with the SAND refinement, called SAND attack trees. To do so, the authors introduce a formal semantics for SAND attack trees, based on *series-parallel graphs* (SP graphs), and extend the bottom-up method for quantitative analysis from classical attack trees formalized in [81] to SAND attack trees.

SAND attack trees considered in [57] use three types of refinements: OR, AND, and SAND. They thus allow to distinguish between actions that can be executed in parallel (connected with AND) from those that need to be executed sequentially (connected with SAND). To formally interpret SAND attack trees, Jhawar et al. use SP graphs. SP graphs are oriented, edge-labeled graphs that contain two distinct nodes – a *source* with no incoming edges, and a *sink* with no outgoing edges – and that can be built in a recursive way from smaller SP graphs, using their *parallel* and *sequential compositions*. The parallel composition glues two SP graphs by identifying their sinks and their sources, respectively. The sequential composition attaches the second SP graph to the first one, by identifying the sink of the first one with the source of the second one. As schematized in Fig. 2, the semantics developed in [57], called the *SP semantics*, interprets an SAND attack tree as a set of SP graphs whose edges correspond to the basic actions of the tree, i.e., its leaves. Each of the leaves of the tree is interpreted as the single edge SP graph, where the edge is labeled with the basic action of the leaf. The parallel and sequential compositions are used to interpret the AND and SAND refinements, respectively. OR refinements are simply interpreted as the union of the sets of SP graphs corresponding to their children. Each SP graph belonging to the set of SP graphs

interpreting a tree corresponds to an attack vector, i.e., a way of achieving the goal of the root node of the tree. Fig. 3 illustrates an example of an SAND attack tree and its SP semantics.

The SP semantics turns out to be a conservative extension of the multiset semantics for classical OR/AND attack trees of [81]. In the multiset semantics, an attack tree is interpreted as a set of multisets, where each multiset is an unordered collection of basic actions representing an attack vector. The SP semantics equips the multisets with a partial order encoding which of the actions need to be performed sequentially.



(a) SAND attack tree (b) and its SP semantics

Fig. 3. The SP interpretation of an SAND attack tree

To allow for an automated treatment of SAND attack trees, the authors of [57] introduce a complete axiomatization of the SP semantics. It is composed of equalities representing semantics-preserving transformations of SAND attack trees. By orienting axioms into a terminating and confluent term rewriting system, the notion of canonical form for SAND attack trees has been introduced and a prototype tool, called SPTool [67], was implemented. SPTool relies on Maude [33] as the underlying computation engine. Its main functionalities are checking the equivalence between two SAND attack trees and computing their canonical forms.

Finally, the notion of attribute domain and the bottom-up algorithm for quantitative analysis, formalized for attack trees in [81], has been extended to SAND attack trees. The attribute domain for SAND attack trees uses (possibly) different combination functions for AND and SAND nodes. This implies that, from a quantitative perspective, actions that can be performed in parallel can be distinguished from those that must be performed sequentially. For instance, if the minimal time attribute is considered, the values of the children of an AND node are combined using max (to reflect a parallel execution mode), and the values of the children of an SAND node are combined using addition (to model the sequential execution mode). This distinction provides more accurate quantitative results compared to the case when all conjunctively connected nodes are treated in the same – either parallel or sequential – way.

The SP semantics was a starting point for the work presented in [54], where more expressive semantics based on linear logic have been developed for SAND attack trees (called causal attack trees in [54]). The reader is referred to Section 3.2 for more details.

3.2 Linear logic interpretation: specialization of attack trees

In [54], Horne et al. develop a framework, based on *linear logic* [43], to interpret classical attack trees, i.e., attack trees with disjunctive (OR) and conjunctive (AND) refinements only, as well as so called *causal attack trees* where the sequential refinement (SAND) is added. The authors show how their framework generalizes existing formalizations of attack trees, based on multisets [81] and SP-graphs [57]. Furthermore, a notion of attack tree *specialization* is introduced. Intuitively speaking, specialization is a relation on attack trees modeling that an attack tree expresses some information more precisely than another one. An example of the specialization of the attack tree from Fig. 1a is given in Fig. 3a where, in addition to the information that both actions a and b need to be executed, it is additionally required that they are executed in this specific order. Specialization generalizes the notion of attack tree equivalence [68] that captures the fact that structurally different

trees represent the same scenario. A relation is then established between the specialization and the quantitative bottom-up analysis of attack trees previously formalized in [81].

The authors of [54] interpret attack trees using the *logical semantics*, based on linear logic and introduced in this paper, which is also used to derive the decision procedures for attack tree specialization, as schematized in Fig. 4. To define the logical semantics for attack trees, Horne et al. use a fragment of linear logic called *multiplicative additive linear logic* (MALL) [53]. They define an embedding of attack trees into positive linear logic propositions, where OR and AND are interpreted using the MALL's additive disjunction and the MALL's multiplicative conjunction, respectively. The relation between the multiset and the logical semantics is established: the inclusion between the multiset semantics of two trees corresponds to the entailment of their corresponding MALL interpretations.

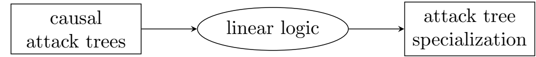


Fig. 4. Linear logic semantics of [54]

The canonical way of equipping a commutative semiring with a preorder[1] is used to define the notion of attack tree specialization. An attack tree t is a specialization of an attack tree t' if, and only if, the tree OR(t, t') is equal to t' modulo the axioms of the algebraic semantics. Since MALL conservatively extends a commutative idempotent semiring, t is a specialization of t' if, and only if, the MALL interpretation of t entails the MALL interpretation of t'. This serves as a basis to define the notion of *soundness* of an ordered attribute domain with respect to the specialization order, ensuring that if t is a specialization of t', then the value quantifying t is smaller than or equal to the value quantifying t'. This generalizes the notion of compatibility between an attribute domain and a semantics [68, 81], ensuring that for attribute domains compatible with an attack tree semantics, equivalent trees (wrt this semantics) yield the same numerical value.

The second part of the paper deals with causal attack trees. The objective of the sequential refinement is to model causal dependencies between attack steps. Contrary to existing work [57] described in Section 3.1, where the conjunctive and sequential refinements are unrelated, Horne et al. consider that, depending on the used applications, one of these refinements specializes the other one. Three semantics are defined for causal attack trees. The first one, called *intermediate semantics*, is based on equivalence classes (wrt particular isomorphisms) of labeled series-parallel graphs. Here, an attack tree is interpreted as a set of series-parallel graphs whose parallel and sequential compositions are used to interpret the AND and the SAND refinements, respectively. This semantics, however, does not accommodate the notion of specialization. Thus, two sets of inequational axioms are introduced to describe that SAND is a specialization of AND, and vice versa. These axioms give rise to two additional denotational semantics: the *ideal semantics* and the *filter semantics*. The ideal semantics maps causal attack trees to ideals. Roughly speaking, an ideal is a set of directed graphs, such that if it contains a graph, then it also contains all the graphs obtained from this graph by adding additional edges. In the ideal semantics, SAND is a specialization of AND. The filter semantics maps causal attack trees to filters. Intuitively, a filter is a set of directed graphs, such that if it contains a graph, then it also contains all the graphs obtained from this graph by removing some edges. In the filter semantics, AND is a specialization of SAND. Horne et al. then prove that the attribute domains for "minimal number of experts" and "time required to make all attacks possible" are sound wrt the filter semantics, whereas the attribute domains for "minimal time required" and "required number of experts on duty to counter any attack" are sound wrt the ideal semantics.

Finally, the MALL logic is extended with a non-commutative operator representing sequentiality. This extension is called MAV (*multiplicative additive system virtual*) [53]. The ideal and the filter

[1]Classically, $x \preceq y$ if, and only if, $x + y = y$, where $+$ is the semiring's additive operator.

semantics are systematized by two fragments of MAV, which yields a method for deciding whether a causal attack tree is a specialization of another one according to one of these semantics. To do so, it suffices to prove that a MAV implication between the two trees is provable.

3.3 Path interpretation: correctness of an attack tree with respect to a system

The goal of the work presented in [15] is to verify the *correctness* of an OR/AND/SAND attack tree with respect to the analyzed system represented as a *transition system*. In this paper, Audinot et al. introduce a novel way of labeling the attack tree nodes and a new semantics for attack trees which is based on paths in the underlying transition system. This allows them to define four correctness properties describing how well the children of an attack tree node refine the node's goal, in the context of a given system. The paper establishes the theoretical complexity of checking the introduced correctness properties. The approach is schematized in Fig. 5.

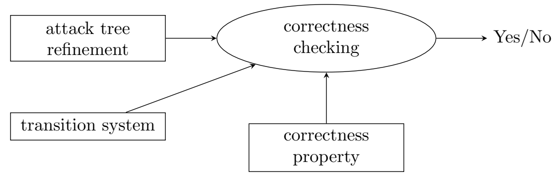


Fig. 5. Correctness checking of [15]

Audinot et al. use transition systems to model real-life systems. A transition system [66] is an operational state-transition model with non–deterministic transitions. In [15], the states of the transition system are labeled with *propositions* that express possible configurations of the real–life system, and the transitions correspond to the actions of the attacker. Attack trees considered in this work make use of the same set of propositions as the underlying transition system. Each node of an attack tree is labeled with a so called *goal*, expressed with the help of two propositions: the *initial configuration* representing the situation before the node's attack starts (preconditions), and the *final configuration*, describing the situation to be reached (postconditions). These pre– and postconditions characterize the states of the transition system from which the attacker can start and where they can end their attack. The nodes' goals are not necessarily independent.

Contrary to the existing formalizations of attack trees, the semantics of the trees considered by Audinot et al. relies on *paths* in the underlying transition system and not on the collection of the attacker's actions. The semantics of a node is defined as a set of paths in the transition system linking a state where the initial configuration of the node's goal is satisfied with a state where the final configuration is valid. The semantics of a disjunctive (OR), conjunctive (AND), and sequential (SAND) composition of nodes is defined using respectively the union, the parallel composition, and the concatenation of the paths belonging to the semantics of its components. For instance, a conjunctive composition of several goals is realized if there is a path that can be decomposed into (possibly overlapping) paths that realize each of these goals. Such a view disallows any kind of parallelism in the execution model.

The correctness of an attack tree refinement is then defined by comparing the semantics of a parent node with the semantics of its refinement, i.e., the semantics of the combination of its children using the parent node's operator. The following four correctness properties are introduced: *meet* – when the intersection between the node's semantics and the semantics of its refinement is non-empty; *under-match* – when the semantics of the refinement is included in the semantics of the parent node; *over-match* – when the semantics of the node is included in the semantics of its refinement;

Table 2. Complexity of correctness checking

	meet	under-match	over-match	match
OR	P	P	P	P
SAND	P	P	P	P
AND	NP-c	co-NP-c	co-NP	co-NP

and *match* when the semantics of a node is equal to the semantics of its refinement. The complexity of verifying the four correctness properties is summarized in Table 2. The verification procedures have been implemented in the ATSyRA Studio tool [3].

Finally, the authors of [17] follow-up on the work initiated in [15] by providing tight bounds for the complexity of deciding the non-emptiness of the path semantics of an attack tree. The non-emptiness problem is shown to be NP-complete for arbitrary attack trees, and NL-complete for attack trees without AND refinements.

4 GENERATION APPROACHES

In practice, attack trees are in general constructed manually by security experts, which leads to several serious drawbacks. First, the manual construction is very *subjective* and depends on the modeler's expertise. This means that trees designed by two different experts for the same system might differ in their size, their structure, and even the attack vectors that they capture. Second, a manual creation process is *tedious* and *error-prone*. Thus, the resulting attack trees may be incomplete i.e., may miss some relevant attack vectors. Third, to facilitate the creation of an attack tree, experts often use libraries of *common attack patterns* or reuse (parts of) the models created in the past for other, similar cases. Starting from existing attack patterns may provide valuable help in the design process, however, it may result in very generic trees that do not properly reflect the subtleties of the analyzed system, which may impact possible attack vectors.

Due to the above-mentioned weaknesses of the manual construction, approaches for (semi-) automated attack tree generation have recently attracted the attention of the security and formal methods communities. Various generation techniques emerged for different kinds of input models. The approach presented in Section 4.1 makes use of a process algebra commonly employed to model *behavior of networks*. Methods described in Section 4.2 and 4.3 are dedicated to the security analysis of *physical* and *socio-technical* systems. The work presented in Section 4.4 is based on the notion of *refinement* expressing possible decompositions of a goal into subgoals. Finally, Section 4.5 presents a method based on the Boolean satisfiability problem (SAT), that aims to *guide* a security expert in designing a pertinent attack tree for a given system. The main features of the approaches described in this section are summarized and compared in Table 3.

Table 3. Generation approaches for attack and attack–defense trees

Sec.	Input model	Output	Connectors	Tool
4.1	Process algebra	Attack tree	OR, AND	Prototype
4.2	Domain-specific model	Attack tree	OR, AND, SAND	ATSyRA [2]
4.3	TREsPASS model	ADTree	OR, AND	TREsPASS tool [10]
4.4	Set of SP graphs and set of refinements	Attack tree	OR, SAND	Theoretical foundations
4.5	Transition system and a partially constructed attack tree	Attack tree	OR, AND, SAND	Theoretical foundations

Methods for automated generation of attack trees usually rely on sophisticated and complex formal engines (e.g., model checking, SAT solving) but once implemented, they act as "push button" solutions assisting the expert in the laborious task of attack tree creation. Thus, from the user's perspective, automated generation is simple and proven, as long as the specification of the system to be analyzed is available and is correct.

4.1 Process algebra-based generation of attack trees

Vigo and Nielson were the first ones to address the problem of automated generation of attack trees, in [105] and [106]. They introduced a procedure, summarized in Fig. 6, that takes a system modeled using the *value-passing quality calculus* [85] and a target location or asset, and generates an AND/OR attack tree representing how the attacker may reach the location or acquire the asset.

Using this procedure, one can generate attack trees covering all possible attacks or only the attacks of minimal cost. Generation of the latter uses an SMT (satisfiability modulo theories) solver.

The value-passing quality calculus is a particular type of process algebra. In the value-passing quality calculus, a system is formalized as a set of processes. Processes can run sequentially or in parallel, and can broadcast and receive messages to and from channels. In addition to a simple reception of a single message, a process can wait for multiple messages at the same time, and continue when some specified subset of these messages has been received. The channels of the value-passing calculus are used to model security checks in the real system. Security checks are measures that protect parts of the system from an attacker. These can for instance be physical objects used to control the access (e.g., keys or badges) or means ensuring the secrecy of data (e.g., passwords). The capacity of the attacker to bypass a security check is modeled by their ability to send a message to the channel representing the security check. For example, sending a message to the channel "password" means knowing the password. Basic actions of the attacker are of two types: the attacker can either gain the ability of sending messages to a channel or actually send a message to a channel, if they already have this ability. Notice that basic actions are not repeatable – once the ability to send a message to a particular channel is obtained, it will never be lost.

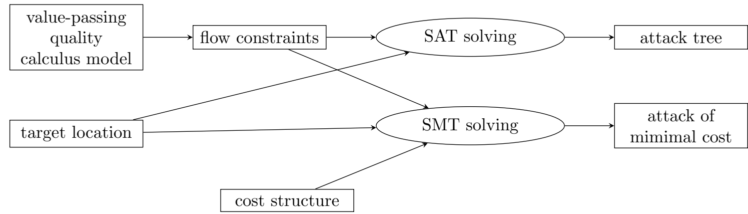


Fig. 6. Attack tree generation by [105]

The security analysis of a system modeled using the value-passing calculus consists in finding attacks and displaying them. This is done in two steps: first, so called *flow constraints* are constructed, and used to generate attacks in the system; then, the attacks are displayed using an attack tree. Flow constraints are propositional formulæ describing *how* the attacker's subgoals can be accomplished. Examples of such subgoals are channels that the attacker can know, variables that they can control, and some locations of interest in processes they can reach. Attacks are generated from flow constraints using a SAT-based approach. First, each flow constraint representing how the attacker can gain access to a channel is replaced by the disjunction between the flow constraint itself and a proposition meaning guessing the channel. This transformation models the ability of the attacker to access the channel. After applying these transformations, a conjunction of the flow constraints is built and can be given to a SAT solver in order to compute all its models. Attacks are then generated from the models by looking only at the valuation of propositions corresponding to channel guesses. The attacks given by the SAT approach are under-approximations of real attacks, in the sense that they only say which channels need to be guessed, but not how many times a message needs to be sent to a single channel.

In the second step, the attack tree is generated from flow constraints. In that context, an attack tree is a propositional formula. When no costs are considered, the attack tree is built from flow constraints by *backward chaining*, as shown in [105]. The idea of the backward chaining algorithm is the following: initially the attack tree is set to be a formula composed of a single positive literal, corresponding to the root goal. As long as there are positive literals in the formula, one of them is selected and each of its occurrences is replaced with the corresponding flow constraint. Attacks are valuations, i.e., assignments of truth values to Boolean variables, satisfying the formula obtained by backward chaining. The attack tree is thus this very same formula. The procedure for the flow

constraints' generation as well as the backward chaining algorithm have been implemented in a prototype tool which uses an external viewer to display the obtained attack trees. More details about the tool can be found in [106].

Cost can also be assigned to basic actions to perform quantitative analysis. Cost structure is defined using a partially ordered set of values and an associative, commutative operation on this set with an identity element. This operation computes the cost of an attack by combining the costs of the basic actions in the attack. The purpose of the quantitative analysis is to generate an attack tree containing only those attacks that have minimal cost. An optimal attack is an optimal solution to the satisfiability of the flow constraints, modulo the SMT theory derived from the cost structure. The SMT solver Z3 [34] is then used to generate valuations corresponding to attacks of minimal cost, as shown in [106]. Multiple attacks can be obtained by successively adding constraints to the query, expressing that an answer cannot be one of the previously found solutions, or that it cannot yield a cost value greater than the previously found optimal solution.

4.2 ATSyRA methodology: generation of attack trees for physical systems

In [89, 90], Pinchinat et al. address the problem of generating an attack tree for a given system whose topology and/or behavior are modeled using a *domain specific language* (DSL). The general idea is to specify the initial state for the attacker (e.g., that he is outside of a military building) and automatically generate an OR/AND/SAND attack tree describing how the attacker can reach a target state (e.g., reach an office where a confidential document is being printed). The generation procedure has been implemented in a tool called ATSyRA– *Attack Tree Synthesis for Risk Analysis* [2]. ATSyRA employs model checking techniques to find possible ways of reaching the target state starting from the initial conditions, and performs parsing and merging to factorize the obtained paths into a humanly understandable attack tree. The ATSyRA methodology is depicted in Fig. 7.

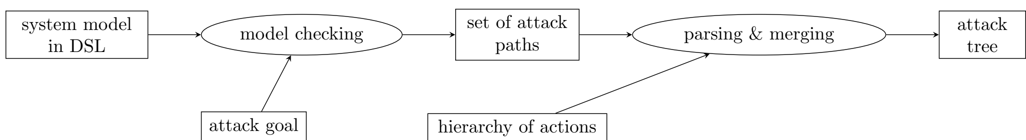


Fig. 7. Attack tree generation with ATSyRA [89, 90]

ATSyRA takes as input a formal description of the analyzed system in a domain specific language. The tool compiles this description into a symbolic transition system specified in the guarded action language (GAL) which is the input language of the ITS-tool model checker [103]. The latter generates a set of attack scenarios expressed as sequences of elementary actions available to the attacker. The OR combination of these attack scenarios forms an OR/SAND attack tree. However, such a flat attack tree usually contains many repetitions, may be very large, and thus difficult to analyze. To address this issue, and transform such a flat attack tree into a *factorized* one having usable internal structure, the authors of [89, 90] use parsing and merging. This transformation relies on the notion of hierarchy of actions. The hierarchy is specified by the rules of an acyclic grammar, where terminals are basic actions available to the attacker, and non-terminals represent high-level actions that may be specified by the user. The objective of high-level actions is to give names to possible meaningful refinements. For instance, instead of using two sequences composed of basic actions "find-PIN" and "steal-card", one may define a high-level action "get-credentials" and specify the rule "get-credentials $\longrightarrow$ AND (find-PIN, steal-card)". The parsing takes care of identifying attack scenarios involving both actions "find-PIN" and "steal-card" which are then factorized using the high-level action "get-credentials". In order to have a meaningful hierarchy of actions, the

rules for high-level actions need to be written by security experts. This implies that the attack tree generation procedure of ATSyRA is semi-automatic. The rules for high-level actions may use OR, AND, and SAND operators, so their application results in factorized OR/AND/SAND attack trees.

ATSyRA has been developed using the *Eclipse Modeling Framework* [2]. The current implementation supports a DSL of physical buildings, that captures components, such as zones, doors, offices, escalators, stairs, as well as items, including keys, badges, alarms, etc. However, it could easily be adapted to other DSLs, e.g., those describing computer networks. To do so, one only needs to define the set of basic actions available to the attacker and specify desired high-level actions and the corresponding rules.

4.3 TREsPASS: generation of attack(–defense) trees for socio-technical systems

In [55, 56], Ivanova et al. automatically generate an AND/OR attack tree from a graph-based model of a socio-technical system, using *recursive policy invalidation*. The objective is to generate an attack tree representing how the considered socio-technical system can be attacked. The approach to the problem of generating attack trees and ADTrees considered in this section has been developed within the EU project TREsPASS[3]. It is one of the steps in the process of analysis of the system's security, performed along the TREsPASS guidelines. The entire TREsPASS approach, together with its philosophy, has been presented by Probst et al., in [93].

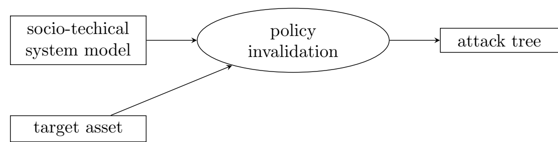


Fig. 8. Attack tree generation by [56]

The graph-based model of the system has nodes representing *locations*, *actors*, *processes*, and *items*. Nodes can contain *assets*, which are items or data. Locations are connected by directed edges that depict how actors or processes can move between them. Conditions in which actions can be performed by actors or processes are defined by *policies*. The latter can be global or bound to specific locations. A policy may require *credentials* which are data, items or predicates. The attack model is the classical model of attack trees, however, the children of conjunctively refined nodes are ordered from left to right during the generation, so the AND nodes can be interpreted as sequential nodes. Finally, subtrees may be repeated in the generated tree.

Attack trees generated in this work depict how an attacker may invalidate a policy in a given system, see Fig. 8. The generation of such an attack tree is done by recursive policy invalidation [63, 64]: starting from a policy to invalidate, find all possible actors who could invalidate the policy, i.e., the potential attackers. Then, identify all the pairs "(action, location)", such that an attacker performing "action" at "location" leads to invalidating the policy. For every potential attacker and for every such a pair "(action, location)", the following steps are performed: first, identify all assets that are needed to perform "action" at "location"; second, generate all paths for the potential attacker in the system to successively obtain the required assets, reach "location" and perform "action"; and third, for each of these paths, identify all the policies that need to be invalidated for the attacker to follow this path. The algorithm recursively constructs an attack tree for every policy, and then combines them using an AND node to get a tree representing a single path. Finally, the trees corresponding to particular paths are combined under a common OR node, resulting in a tree invalidating the initial policy. The algorithm is detailed in [56].

The approach developed in [55] is adapted by Gadyatskaya to generation of ADTrees, in [38]. The method presented in [38] relies on a simplified model of a socio-technical system, in which nodes

[2]https://www.eclipse.org/modeling/emf/
[3]https://www.trespass-project.eu/

represent items, including, *infrastructure locations*, *actors*, and *objects*. If an item i is accessible from an item i', then there exists a directed edge from the node representing i to the node representing i' in the system model. The only local policies taken into account are the ones specifying which credentials (physical objects or data) are required to access a particular item from another one. The policies are additionally equipped with information on the mechanism that enforces them, e.g., a policy stating that accessing a room requires a badge might be enforced by an RFID reader.

Below, we give an overview of the generation process of an ADTree, for a given system and a goal of accessing an item i. The first step is to automatically exploit the model of the system and the local policies: if i is accessible from an item i', then among the child nodes of the root node there is a node labeled "access i from i'". If accessing i from i' requires satisfying a local policy p, a countermeasure corresponding to p is attached to this node. This countermeasure can be again countered by the attacker, in two ways: either by satisfying the policy or by breaking the enforcement mechanism of p. Satisfying the policy requires obtaining appropriate credentials, i.e., accessing some other items. The subtrees for accessing those items are created in the same manner. The second, semi-automatic step relies on the presence of a human analyst. Generic countermeasures are attached to some of the nodes of the attacker. It is up to the analyst to decide whether or not they are applicable in the system under consideration, and if they are, to specify them further.

4.4 Biclique problem for a refinement-aware creation of attack trees

A theoretical framework addressing the problem of attack tree generation has been proposed by Gadyatskaya et al., in [42]. This work has two main contributions. First, given a semantics for attack trees and a set of allowed refinements, the authors formally define the *attack-tree generation problem*. They provide a solution to this problem in the case of the SP semantics, introduced in [57] and discussed in Section 3.1. Second, Gadyatskaya et al. propose an approach to generate an attack tree for a given system represented with the help of a *labeled transition system* (LTS). The particularity of the obtained tree is that it is *refinement-aware*, i.e., that its nodes correspond to the meaningful levels of abstraction that can be expressed using the underlying LTS components.

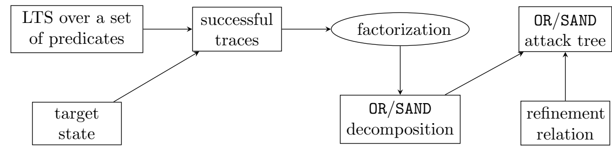


Fig. 9. Attack tree generation by [42]

The formalization of the attack tree generation problem relies on the notion of refinement. Given a set of actions $\mathbb{B}$, a *refinement* is an expression of the form $b \triangleleft \mathtt{OP}(b_1, \ldots, b_n)$, where $\mathtt{OP} \in \{\mathtt{OR}, \mathtt{AND}, \mathtt{SAND}\}$, and $b, b_i \in \mathbb{B}$. The attack tree generation problem is defined wrt a given attack tree semantics, understood as an equivalence relation on the set of all attack trees, as in [57, 68, 81]. The input to this problem is the interpretation of an attack tree in a given semantics and a set of refinements. The output is an attack tree having the same semantical interpretation as the input, where the nodes are labeled by the elements of $\mathbb{B}$, and such that all refinements in the tree belong to the set of refinements provided by the input.

Gadyatskaya et al. develop an algorithm that provides a solution to the attack tree generation problem in the case of the SP semantics [57] which interprets attack trees as sets of series-parallel graphs (SP graphs). This algorithm relies on the *edge biclique problem* which is known to be NP-complete [87]. The proposed solution generates attack trees with OR and SAND refinements only.

Finally, the authors of [42] also show how to generate an OR/SAND attack tree for a given system modeled with the help of an LTS. Recall that an LTS is composed of a set of states containing the initial state, a set of labels, and a binary relation defining possible transitions between two states,

each transition having a label. In this work, every state is represented with the set of predicates valid in this state. The labels of the generated attack tree nodes are taken from the set of system states. To define the set of refinements, which is one of the inputs to the attack tree generation problem, the abstraction relation on the set of states is introduced. State s is said to be more abstract than state s', denoted $s \sqsubseteq s'$, if all predicates of s also belong to s' (therefore, this relation is a partial order). Given the abstraction relation on the set of states, the abstraction-based set of refinements is defined as the smallest set containing $s \triangleleft \mathsf{OR}(s_1, \ldots, s_n)$, whenever $s \sqsubseteq s_i$, for $i \in \{1, \ldots, n\}$, and $s \triangleleft \mathsf{SAND}(s_1, \ldots, s_n)$, whenever $s \sqsubseteq s_n$. This means that the goal of an OR node must be more abstract than the goals of all of its children and the goal of an SAND node must be more abstract than the goal of its right-most child. Given an LTS and an abstraction-based set of refinements defined as above, an attack tree is then generated from a set of successful traces in the LTS, i.e., traces that start from the initial state and end in any state containing a desired (set of) predicate(s). A predicate of interest can for instance state that an attacker learned a secret or reached a specific location in the system. The traces of the LTS indicate sequences of transitions that are valid in the modeled system and can thus be connected with an SAND operator. Thanks to a factorization, the OR/SAND skeleton of the attack tree (i.e., an attack tree with no labels at the refined nodes) is obtained. To label the refined nodes of the skeleton, the abstraction-based set of refinements is used, as schematized in Fig. 9.

The idea behind the model of Gadyatskaya et al. is similar to the one from [15] described in Section 3.3. Both approaches use LTSs to model the analyzed system, and attack trees are interpreted with sets of paths (traces) in this LTS. The attack tree generation procedure developed in [42], relying on an abstraction-based set of refinements, implies that all successful traces corresponding to a node's children level are included in the set of successful traces of the node. This means that, in the spirit of the refinement properties defined in [15], attack trees generated by Gadyatskaya et al. satisfy the global *under-match* property.

4.5 Guided design of attack trees by tracking useful positions

An orthogonal approach to generate attack trees has been proposed by Audinot et al. in [16]. The contribution of this work is to guide an expert in a *manual design* of attack trees by exhibiting leaves that are worth developing as they take part in an optimal attack of the system. The approach is based on automata construction and makes use of the propositional satisfiability problem.

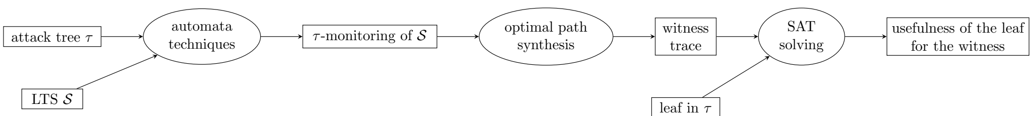


Fig. 10. Guided design of attack trees by [16]

The framework developed in [16] captures attack trees with OR, AND, and SAND refinements. Their leaves are labeled with reachability objectives by means of postconditions expressed as atomic propositions. Such trees have a natural regular language semantics of traces, with an effective construction of the finite-state automaton recognizing this language. The real-life system to be analyzed is modeled as a labeled transition system (LTS) with quantitative features (typically, a priced timed automaton [21]), and whose states are decorated by atomic propositions. By synchronizing the automaton and the LTS (with a standard product in the spirit of what is used to synchronize a Büchi automaton with an LTS in model checking), all traces of the resulting LTS belong to the attack tree trace language. Relying on well-known techniques to synthesize an optimal trace in an LTS, an optimal attack of the system can be obtained. Next, a reduction to the propositional logic

satisfiability problem allows to identify leaves of the tree that may be involved in such an optimal attack. As a result, the expert creating the tree is advised on which leaves in the partially-refined tree may contribute to this attack and are therefore worth being refined further. Fig. 10 illustrates the entire process.

More technically, to make a link between the attack tree and the LTS, a common alphabet is used: this alphabet is the powerset of some set Prop of atomic propositions. Each state of the LTS $\mathcal{S}$ is assigned with a subset of Prop (an element of 2^{Prop}), modeling propositions valid in that state. A path in $\mathcal{S}$ is abstracted as a *trace* by keeping only the sequence of propositions that hold along this path. Each leaf of an attack tree is labeled with some atomic proposition $\gamma \in \mathrm{Prop}$. The internal nodes of attack trees are labeled with their refinement type (OR, AND, or SAND).

Formally, an attack tree leaf corresponds to linear-time reachability property in $\mathcal{S}$, which is formalized using *trace semantics*. Given an attack tree, its trace semantics is a language over the set of positive conjunctive Boolean formulæ composed from the elements of Prop, as follows: the semantics of a leaf node labeled by $\gamma \in \mathrm{Prop}$ contains all the words over 2^{Prop} that end with element γ in their last letter; the semantics of an OR node is the union of the semantics of its children; the semantics of the SAND and AND nodes is defined using an *enhanced concatenation* and an *enhanced shuffle* of languages, respectively. Intuitively, the enhanced concatenation models a sequential composition of the attacker's goals: either the goals are achieved one after another (standard concatenation), or the goals are achieved simultaneously. The shuffle captures the achievement of several goals in any possible order, possibly simultaneously. Audinot et al. show that the trace semantics of an attack tree τ is a regular language and that an automaton, denoted $\mathcal{A}_\tau$, accepting this language can be effectively constructed.

Automaton $\mathcal{A}_\tau$ is used to monitor $\mathcal{S}$, otherwise said, to restrict the system's traces to the trace semantics of attack tree τ. This is obtained as a product of $\mathcal{A}_\tau$ and $\mathcal{S}$, called *τ-monitoring of $\mathcal{S}$*, and denoted by $\tau[\mathcal{S}]$. The main contribution of this work is to take advantage of the monitored system $\tau[\mathcal{S}]$ in order to perform an early-stage quantitative analysis, even if the attack tree is not fully deployed yet, and to guide the expert in further refinement of the tree. Typically, if the system model is a priced timed automaton, and according to the state of the art results, the problem of reaching a final state in $\tau[\mathcal{S}]$ in, for instance, a cost-optimal way is decidable [4, 84], the corresponding optimal path can be synthesized. Thanks to the way the monitored system $\tau[\mathcal{S}]$ is constructed, this path reflects an attack on $\mathcal{S}$, covered by the attack tree τ, and its trace (in the semantics of τ by construction) is called a *witness*. The cost of this witness gives a faithful value of the partially-refined attack tree.

Once the witness is known, the aim is to identify which positions (and of uttermost importance which leaf positions) in the tree τ justify the membership of this witness in the trace semantics of τ, so that the security expert can be advised on which leaves of his partially-refined attack tree are relevant for this witness and should therefore be deployed further. This is achieved by verifying satisfiability of an appropriately constructed propositional formula.

The framework presented in this section has several similarities to the one introduced in [15] and described in Section 3.3. Both approaches use an operation model (an LTS) of the analyzed system while creating or analyzing an attack tree. In both cases, the labels of attack trees correspond to the reachability goals in the underlying LTS and not, as in the majority of classical frameworks for attack trees, to basic actions composing the attacks described by the tree. However, in contrast to [15], the reachability goals considered in [16] represent only the postconditions to be reached by the attacker, i.e., the initial preconditions are not explicitly specified. This implies that the operation of enhanced shuffle, formalizing the semantics of the AND nodes, is associative (contrary to the parallel composition of paths employed in [15]), and thus the considerations of [16] can be reduced

to binary (instead of unranked) trees. Finally, unlike in [15], the internal nodes of attack trees in [16] are labeled with the type of their refinement and not with extra reachability goals.

5 STATIC ANALYSIS

Classical problems one can consider given an attack(–defense) tree involve determining quantitative properties of the modeled scenario, e.g., minimal cost or minimal time needed for achieving the root goal, maximal probability of the root goal being achieved, etc. The standard approach to solve these problems is based on a simple *bottom-up algorithm*, described first by Schneier in [101] and then formalized for attack trees in [81] and for attack–defense trees in [68]. In this classical approach, the expert estimates the input values quantifying the property of interest at the non-refined nodes (i.e., basic actions). The values for the remaining nodes are then computed in a bottom-up way, depending on the type of the refinement of the node. While computationally fast – linear in the number of nodes in the tree – this bottom-up procedure has two main drawbacks: it takes *only one quantitative property* into account at a time and it assumes that all nodes in the tree are *independent*.

In this section, we present formal frameworks that improve the classical bottom-up approach and lift some of its limitations. First, we focus on attack–defense trees with independent nodes. In Section 5.1, an approach based on Pareto frontier for *multi-objective* quantitative evaluation is described. Section 5.2 discusses how to make use of integer linear programming to tackle the problem of *optimal* allocation of defender's resources. The remaining two sections present the frameworks where a special type of dependencies between the nodes is considered, namely where *several nodes may carry the same label*, i.e., represent exactly the same action. An interesting approach for approximating the minimal cost of an attack vector in attack trees with repeated leaves is described in Section 5.3. The work presented in Section 5.4 shows how to extend the classical bottom-up algorithm to attack–defense trees with repeated basic actions.

In contrast to the methods covered by Section 6 and 7, the current section concentrates on static approaches that disregard the order or time in which actions are executed by the actors. A comparative view for the approaches described in Section 5, 6, and 7 is given in Table 4.

5.1 Pareto efficient strategies in attack–defense trees

In [11], Aslanyan and Nielson provide a formal approach to the problem of *multiple parameter optimization* in ADTrees. Every set of basic actions of the actors (called *strategy* throughout this section) is assigned a vector $v = (v_1, \ldots, v_k)$ of $k \geq 1$ values. Some of the values might represent costs associated with execution of the actions of the root player that belong to a given strategy. Among them there might also be the probability of the root goal being achieved when the strategy is executed (*probability of success*). The aim is to determine the strategies that achieve the root goal and *optimize all of the values at once*. Such optimal strategies are defined in terms of *Pareto efficiency*, see Fig. 11. A strategy is optimal if its corresponding vector v is Pareto efficient in the set of vectors corresponding to all strategies, i.e., if every other vector that offers an improvement wrt v on at least one coordinate entails a worsening on some other coordinate.

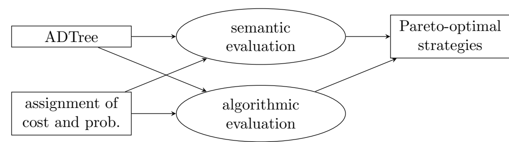


Fig. 11. Pareto efficient strategies by [11]

The underlying assumption of the whole framework is the independence of basic actions performed by the actors. The main focus is put on the class of so called *linear trees*, i.e., trees where every action appears exactly once. The basic model of ADTrees introduced in [68] is extended with a negation operator, which allows for capturing the situation in which execution of an action by an

actor makes it impossible for them to perform some other action. Aslanyan and Nielson use this operator also for defining a specific class of ADTrees, called *polarity–consistent trees* (PCTrees), in which multiple occurrences of basic actions are allowed under some constraints.

Each of the basic actions is assigned two probability values: a probability of achieving the goal it represents in the case of attempted execution, and a probability of achieving the goal in the case when the action is not executed (in the Boolean case, where the problem of satisfiability of the root goal is tackled, these values are 1 and 0, respectively). Furthermore, each of the actions is decorated with a vector $c = (c_1, \ldots, c_m)$ of $m \geq 0$ real-valued costs. In this setting, two approaches to the problem of determining Pareto optimal strategies that maximize the probability of success and minimize costs are considered. In the first one, called *semantic evaluation*, the probabilities and costs corresponding to all possible strategies (with the cost of a strategy being a coordinate-wise sum of costs of the actions that constitute the strategy) are computed, and only then the Pareto optimal values are selected. This method has the drawback of high complexity, due to the fact that the number of strategies in an ADTree is exponential in the size of the tree. To overcome this difficulty, the authors of [11] develop an alternative method, called *algorithmic evaluation*. For the case when $m = 0$, this method is a combination of two standard bottom–up procedures, and determines the lowest and the highest values of probability of success in a linear tree, in the time linear in the size of the tree. Boolean version of this problem is solved similarly in the class of PCTrees. In the Boolean variant the result reflects the influence of the actions of the other actor on the actions of the root actor. For instance, the result can highlight the fact that the root goal is always achieved, no matter what the other actor does, or that the other actor can select actions that ensure that the root goal cannot be achieved by the root actor. The algorithmic evaluation method in the case of $m = 1$, that is, in the presence of both probability and a single cost, propagates up to the root of a tree only the Pareto efficient values. In a linear tree, the result obtained at the root coincides with the result of the semantic evaluation, and, again, is obtained in the time linear in the size of the tree. It is worth noticing that the complexity of the algorithmic evaluation increases with the growth of the number m of costs associated with the basic actions, i.e., for any fixed m there is an ADTree T of size linear in m and with the number of unique Pareto optimal strategies exponential in the number of nodes of T. The computation of the set of Pareto optimal solutions for the probability and cost parameters has been automated in the Attack Tree Evaluator tool (ATE) [9, 10].

In the framework of [11] the possible behavior of the actors is described by sets of actions that they execute. This description does not take the order of the actions' execution into account. To additionally capture the order of execution of actions, Aslanyan et al. develop a framework based on stochastic two-player games, in [13] (see Section 7.3). Contrary to the approaches for multi-parameter optimization in ADTrees based on timed automata (cf. Section 6.2), the methods presented in [11] do not capture the possibility of a single action being executed multiple times.

5.2 Selection of an optimal set of countermeasures using integer linear programming

The work of Kordy and Wideł presented in [72] focuses on advising the defender how to distribute available funds among possible countermeasures. Given the budget of the defender and an AND/OR ADTree decorated with costs of execution of the basic actions of the actors, the aim is to select a set of basic actions the defender can afford, that optimizes some value, e.g., maximizes the necessary investment of the attacker in achieving the root goal. Optimization problems are formulated in terms of *integer linear programming* (ILP) [24]. The approach is depicted in Fig. 12. It assumes that there are no repeated basic actions in the ADTree under consideration.

To identify the attacker's strategies and ways of their prevention, a novel semantics for ADTrees is developed. Under this semantics, called *defense semantics*, ADTrees are interpreted as sets of

pairs of the form (A, D), where A and D are sets of basic actions of the attacker and the defender, respectively. The meaning of the pair (A, D) belonging to the defense semantics of an ADTree is the following: the set A describes a rational strategy of the attacker, i.e., there exists a (possibly empty) set D' of basic actions of the defender such that A is a minimal (wrt the inclusion) set of basic actions execution of which achieves the root goal if the countermeasures implemented by the defender are exactly those from D'. The set D is a minimal set of countermeasures implementation of which makes A unfeasible. Therefore, the defense semantics provides the defender with the information on possible behavior of a rational attacker, and on possible ways of countering such a behavior (note that the attacker's strategies that cannot be countered are not kept in the semantics). Given the defense semantics, the defender has an incentive to determine a set of countermeasures they can afford, implementation of which optimizes some value. The optimization problems considered in [72] include: maximization of the number of unfeasible attacker's strategies; maximization of the investment of the attacker necessary to achieve the root goal; and minimization of the impact originating from feasible attacker's strategies. Nevertheless, the general structure of the functions to be optimized makes the formulation of other optimization problems possible as well.

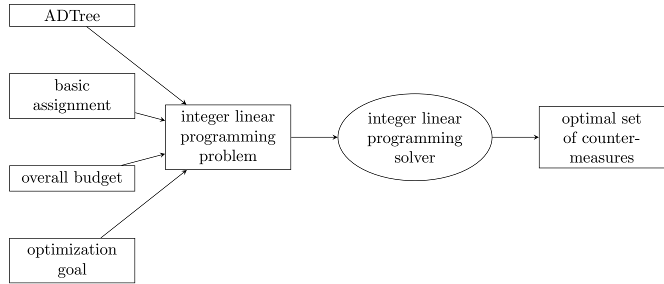


Fig. 12. Linear programing for ADTrees by [72]

Kordy and Wideł encode the relations between the basic actions executed by the defender and the elements of the defense semantics using appropriate Boolean variables and linear inequalities. Together with a linear function of the above-mentioned variables to be optimized (e.g., describing the necessary investment of the attacker) this system of inequalities constitutes an ILP problem. The authors of [72] implement a prototype tool which translates an ADTree decorated with costs into a specification of a relevant ILP problem accepted as input by a free ILP solver LP_SOLVE [23]. The specification is passed to LP_SOLVE and the set of countermeasures implementation of which optimizes the given function is returned.

5.3 Efficient approximation of the cost of a cheapest attack

The focus of Buldas et al. in [30] is to provide proofs that for some attack trees no profitable attack vectors exist. Formally, the problem is addressed by determining whether the cost of a cheapest attack is greater than a given threshold. This is partially achieved by evaluating a lower bound for the cost of a cheapest attack via a combination of a *weight reduction* technique and a standard bottom-up procedure, as illustrated in Fig. 13.

This work considers standard AND/OR attack trees with possibly repeated basic actions. Attack trees are modeled with *monotone Boolean functions* over propositional variables representing successful executions of particular basic actions by the attacker. An attack in a tree is a minterm of the corresponding formula, i.e., a conjunction of some of the variables that implies the truth of the whole formula. Given a weight function w that assigns non-negative, real values to the propositional variables, the cost of an attack is the sum of weights of its variables. For a tree Φ, a weight function w, and a profit threshold K, the aim is to determine whether it is profitable for the

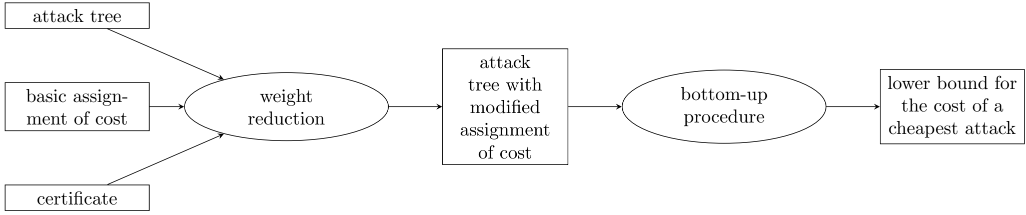


Fig. 13. Approximation of the minimal cost of an attack by [30]

attacker to execute an attack, i.e., whether the weight of a cheapest attack in Φ, denoted with $w(\Phi)$, does not exceed K. This problem can be formulated in terms of the *weighted monotone satisfiability problem*, which is known to be NP-complete [30]. To bypass the complexity of this problem, the authors of [30] propose a method for computing a lower bound for $w(\Phi)$, which is then compared with K. The quality of the obtained lower bound is indicated by the relative error of the method, i.e., by the ratio of the difference between the upper bound and the lower bound to the lower bound.

The lower bound for $w(\Phi)$ is obtained in two steps. First, a weight reduction technique is employed. For every propositional variable x that appears multiple times in the formula Φ, each of its occurrences is replaced with a new variable, and the weight of x is distributed among the new variables, i.e., the sum of weights of the new variables is equal to $w(x)$. Information on how the weights of repeated variables of Φ should be distributed among their occurrences is called a *certificate* for Φ. In the propositional formula obtained after this step, every variable appears exactly once. The exact cost of the cheapest attack in this new tree can be therefore evaluated by the classical bottom-up procedure, in which the values assigned to the leaf nodes are propagated up to the root of the tree, using the sum at the AND nodes and the minimum at the OR nodes. This exact cost is computed in the second step of the method. It provides a lower bound for $w(\Phi)$. Furthermore, Buldas et al. prove that if in every subformula of the form $G \wedge F$ of Φ the subformalæ G and F have at most one variable in common, then this lower bound is actually equal to $w(\Phi)$. If the lower bound is greater than the profit K, then it is not profitable for the attacker to conduct an attack.

Once a certificate for Φ is known, it is computationally easy to verify it, that is, to check whether the lower bound for the cost of the cheapest attack in Φ that it provides exceeds K. The choice of a certificate that would achieve the best approximation of $w(\Phi)$ remains problematic. It is worth noting that the exact value of $w(\Phi)$ can be obtained using methods of [73] in a time linear in the number of nodes of a tree and exponential in the number of repeated basic actions (cf. Section 5.4).

5.4 Quantitative analysis of attack–defense trees with repeated actions

It is known that if a tree contains repeated basic actions, the standard bottom-up procedure for quantitative analysis of attack(–defense) trees might return distorted results. This issue is tackled by Kordy and Wideł in [73], in the case of standard AND/OR ADTrees. The authors investigate conditions ensuring that the result of the bottom-up procedure performed in the presence of repeated basic actions is correct, and develop an alternative evaluation method in the case when these conditions are not satisfied, as schematized in Fig. 14.

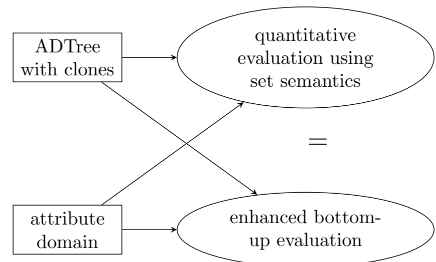


Fig. 14. Quantifying ADTrees with clones by [73]

In [73], two nodes labeled with the same basic action represent exactly the same single instance of the action. Such nodes are called *clones*. To interpret ADTrees with clones, this work uses the *set semantics* introduced in [27]. Under the set semantics interpretation, an ADTree is a set of pairs (A, D), where A is a set of basic actions of the attacker and D is a set of basic actions of the defender, such that if all of the actions from A are executed, and none of the actions from D is executed, then the root goal is achieved.

Two approaches for evaluating quantities (called attributes) on ADTrees with clones are presented. Both of them start with estimating input values for all basic actions. In the first approach, the attribute is evaluated directly on the tree by using the standard bottom-up algorithm. In the second one, the set semantics of the tree is constructed, and the attribute value corresponding to the tree is obtained by combining the values of basic actions composing the pairs (A, D). While the first of the two methods has complexity linear in the number of nodes in the tree, it might fail to provide the correct result, because the values of the clones might be counted several times. The second approach returns a value that indeed corresponds to the aspect of the scenario under consideration, but requires constructing the semantics, which might be computationally expensive. Therefore, it is desirable to ensure that the fast method returns the correct result.

Both of the methods drafted above can be formalized in terms of *attribute domains* [68, 81]. The authors of [73] focus on attribute domains induced by idempotent commutative semirings $(D, \oplus, \otimes)$. They prove that if there are no clones in a tree, or if both operations of the underlying semiring are idempotent, then the two methods yield the same result. This means that, under the above assumptions, the correct result of the evaluation on the set semantics can be obtained with the fast bottom-up procedure.

Kordy and Wideł also identify a condition that, for the case when the tree contains clones or the $\oplus$ operation is not idempotent, allows for an alternative method for evaluation of attributes that might be computationally less expensive than constructing the set semantics. The idea behind the method is to first determine which of the repeated basic actions of the attacker appear in every set A such that the pair $(A, \emptyset)$[4] is an element of the set semantics of the tree. Such a basic action is called *necessary clone* and can be recognized in time linear in the number of nodes of the tree. The remaining repeated basic actions are called *optional clones*. First, all of the necessary clones are assigned value equal to the neutral element for $\otimes$. Then, for every subset of the optional clones, the values assigned to the optional clones are temporarily modified (some of them are assigned the neutral, and some of them the absorbing element for $\otimes$) and the bottom-up procedure is performed. Intuitively, due to this modified assignment of values, some of the optional clones are ignored by this bottom-up procedure, while others are selected whenever possible. The outcomes of all these bottom-up procedures are eventually combined and the result is modified in a way that ensures that the original values assigned to clones are taken into account exactly once.

The time complexity of the above method is linear in the size of a tree and exponential in the number of clones. It is suitable in particular for evaluating the minimal cost of an attack, and therefore for solving instances of the *weighted monotone satisfiability problem* (cf. Section 5.3).

6 TIMED AUTOMATA-BASED ANALYSIS

While being able to provide a security expert with valuable information, the analysis methods presented in the previous section share common limitations. In particular, they do not take into account the temporal dependencies between attack steps or the capability of the actors to attempt to execute a single action multiple times. Interpreting attack(–defense) trees using *timed automata*

[4] The pairs of the form $(A, \emptyset)$ represent scenarios in which the attacker wins no matter what the defender does.

does not only allow to lift these limitations, but also enables employment of the state of the art model checking tools for the purpose of the quantitative analysis.

This section focuses on approaches based on timed automata. A way of interpreting attack trees using *priced timed automata* and an application of such interpretation to the multi-objective quantitative evaluation of attacks is described in Section 6.1. In Section 6.2, an approach for analysis of attack–defense trees using *stochastic timed automata*, allowing to take cost, time, and probability into account simultaneously, is presented. Finally, Section 6.3 is devoted to *attack–defense diagrams*, an expressive model for analyzing attack–defense scenarios that can be seen as an extension of attack–defense trees.

6.1 Attack tree analysis with priced timed automata

In [75], Kumar et al. develop a framework for multi-objective quantitative analysis of attack trees, exploiting the model checking techniques. As illustrated in Fig. 15, attack trees are transformed into networks of *priced timed automata* (PTA) [21] which are then given to the UPPAAL CORA model checker [1] where they are queried for quantitative properties of interest. The objective is to provide an effective way of computing the necessary resources, e.g., time, skills, etc., and the corresponding attack paths leading to the achievement of the root goal of the attack tree. The method also allows to rank the attack paths according to a given quantitative criterion, for instance identify ten cheapest attack paths.

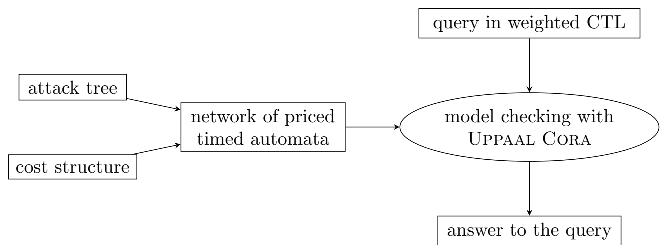


Fig. 15. PTA-based analysis of attack trees by [75]

The model of attack trees considered in this work contains classical AND and OR nodes (called gates) as well as temporal gates SAND and SOR. The attack tree leaves are augmented with structures of costs (e.g., time, skills, resources, damage, difficulty) associated with execution of basic actions (also called basic attack steps) that they represent. Basic actions are supposed to be non-repeatable, i.e., once executed, a basic action is considered successful if encountered later in the tree. Shared subtrees are allowed to reduce the size of the tree, and are treated as if the subtree was actually duplicated at the positions where it appears in the tree.

The authors of [75] translate attack trees into priced timed automata – a quantitative model that combines clocks (to model time, as in timed automata) with a cost function assigning costs to locations and actions of the automaton. Basic actions and gates of the attack tree are translated into a network of PTAs. The UPPAAL CORA model checker [22] – an extension of UPPAAL [78] with cost – is then employed to transform the PTAs' network into a single PTA, by an operation called *parallel composition* of PTAs. Parallel composition synchronizes the transitions of the individual PTAs via joint signals: when the PTA of a child node sends a signal, it triggers a transition in the PTA of its parent node. Quantitative properties to be checked on the tree are expressed using *weighted CTL* [29] that extends the branching temporal logic CTL with costs. The verification of the properties is ensured by UPPAAL CORA allowing to check formulæ of weighted CTL. Optimal attacks can be computed if all objective values but one are fixed. Pareto frontier can also be computed for multi-objective optimization. Finally, the PTA-based analysis of attack trees has been integrated into the ATTOP platform [76].

In contrast to [39] (cf. Section 6.2) where timed automata have been used to analyze ADTrees, the work presented in [75] does not model the defender explicitly, nor does it cover probability.

6.2 Attack–defense tree analysis with timed automata

The main goal of the framework developed in [39] is to model temporal behavior of the attacker in an ADTree and to exploit this modeling for the purpose of quantitative analysis of the underlying attack–defense scenario. Gadyatskaya et al. propose a way of encoding the actors and their basic actions as *networks of timed automata* [5]. Such a network is then provided as input to the Uppaal model checker [20, 78], which allows for extracting strategies of the actors satisfying particular properties, as schematized in Fig. 16. The standard model of ADTrees with OR and AND refinements only is considered. The success or failure of the attacker is seen as a result of the evaluation of the propositional formula corresponding to the tree.

First, an ADTree is used to derive a directed labeled graph, called by the authors of [39] an *attack–defense graph*. This graph represents possible realizations of the scenario modeled by the tree, i.e., combinations of all sets of actions executed by the defender with all potential sequences of the actions executed by the attacker. The attack–defense graph is used to define the attacker's profile, which models the capabilities (what are the actions that the attacker can execute and what are the properties of their execution times) and preferences (the probability that a given action is chosen) of the attacker in any situation that can occur in the scenario. Formally, the attacker is modeled as a *timed transition system* [50] equipped with a description of its non-deterministic behavior. The attack–defense graph and the profile of the stochastic attacker are combined to create a stochastic timed transition system that models possible realizations of the scenario. Given a set of actions executed by the defender, and taking into account the stochasticity of the attacker, the probability of successful execution of basic actions, and the cost of attempting their execution, Gadyatskaya et al. derive explicit formulæ for the probability of the attacker's success, and the expected cost within a given time bound. This naturally leads to the problem of choosing the attacker's profile that optimizes these values.

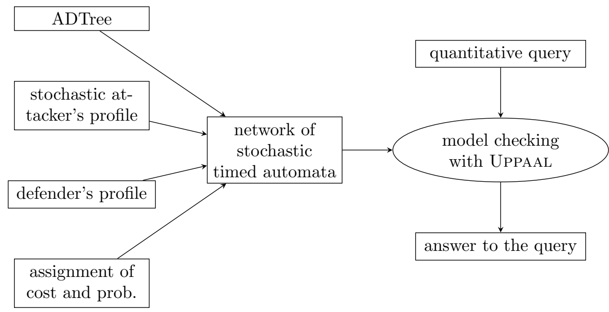


Fig. 16. Uppaal-based analysis of ADTrees by [39] and [47]

The final transition system is encoded using network of *stochastic timed automata*, in a way that ensures that the runs of the network correspond to sequences of transitions in the system. The encoding is performed in a modular manner, i.e., the network consists of an automaton that models the attacker, an automaton modeling the defender, and an automaton for each of the basic actions of the attacker, that models possible outcomes of executing the action. The authors of [39] implemented the encoding procedure, and the implementation outputs a specification of the network that is accepted as input by the Uppaal model checking engine [20], [78]. Using Uppaal, it is then possible to, e.g., determine the probability of a successful attack or the expected cost of succeeding (for a specific attacker profile) within a given time bound.

The approach from [39] is expanded upon by Hansen et al., in [47], with three novelties. First, a dependency between the total cost of execution of an action and the time spent on the execution of the latter is introduced. Instead of being equipped with a real value of cost, as in [39], every

basic action in [47] is assigned a relative cost of execution per time unit. Second, Hansen et al. formalize a profile of a cost-preserving attacker. The probability of a given action being executed by a cost-preserving attacker depends on the relative cost of the action and the maximal possible time needed for its execution. The lower the impact of the execution of an action on the attacker's budget, the more likely the attacker is to execute the action. Since a cost-preserving attacker might not behave in a way that maximizes the probability of success, a parametrization of such an attacker is proposed. In the case of the parametrized cost-preserving attacker, the probabilities based on the impact of the execution of an action on the attacker's budget are additionally weighted. Finally, a method for selecting a configuration of parameters that minimize the expected cost of an attack in a given ADTree and under given stochastic defender is proposed. For a given set of configurations of parameters, a number of simulations of the attack–defense scenario is performed for each of the configurations, and the results (costs of success) are subject to analysis of variance. As long as the analysis of the variance detects differences between the sets of results, some of the configurations are being removed, additional simulations are performed for the remaining configurations, and the results are tested again. When no differences are detected, the results of the simulations are assumed to originate from identically distributed random variables. In particular, it is assumed that all of the remaining configurations of the parameters yield the same (optimal) expected cost of the attacker being successful within the given time bound.

In order for the results of the analysis proposed in both [39] and [47] to be meaningful, the underlying ADTree should satisfy some properties, which seem to be implicitly assumed. Computations of the probability of the attacker's success rely on the assumption of mutual independence of all basic actions. Furthermore, if the actions under an AND node of the attacker can be executed in parallel, this information is lost in the automata interpretation of the tree, since the final behavior of the attacker is represented as a sequence of actions. Finally, it is assumed that every action of the attacker can be executed an unbounded number of times, until it is completed successfully.

6.3 Attack–defense diagram's analysis with stochastic timed automata

Being aware of drawbacks and limitations of the original ADTree model, in particular its inherent inability for capturing dynamic aspects of evolving scenarios, Hermanns et al. proposed a model called *attack–defense diagrams* (ADDs) [51]. While essentially being directed graphs with labeled nodes and arcs, ADDs offer an impressive expressive power due to the number of new ways the basic actions can interact with each other (not only by achieving goals through AND and OR refinements or countering goals of the other actor). This, however, comes at a cost of increased complexity.

ADDs extend ADTrees in several dimensions. Rather than trees, they are directed graphs that admit cycles. Their basic components are *basic events* (instead of classically used basic actions) which represent not only actions that the actors can perform, but also time-driven events that might occur independently of the actors. Single root node representing the main goal of a scenario in an ADTree is replaced with *two sink nodes* (nodes with no outgoing edges), each of them corresponding to the main goal of one of the two actors. In a particular realization of the scenario, the actors perform actions, thus changing their status from *undefined* to *true* or *false* (depending on whether the action is executed successfully or not). These logical values are propagated throughout the diagram via the refined nodes. The result of a realization of the scenario is established when one of the sink nodes changes its status to *true* (the actor whose goal is represented by this sink wins) or *false* (the other actor wins). If both sinks become *true* at the same time, then the result is a draw.

In addition to AND and OR, ADDs support eight new types of nodes: COST, IF, RE, TR, SAND, SOR, NOT, SWP. These include two conditional gates: COST that propagates the *true* values only if the

amount of one of the resources spent by the attacker so far satisfies a specified bound; and IF that propagates the *true* values coming from inputs only if a specific condition is satisfied. The possibility of multiple executions of a single event or causal and conditional dependencies between events can be modeled with the reset (RE) and trigger (TR) gates. To capture sequential behavior, one can use the SAND and SOR gates. Finally, NOT and SWP swap between *true* and *false* and between *true* and *undefined*, respectively.

The framework proposed by Hermanns et al. is sketched in Fig. 17. Basic events are decorated with three attributes: cost of attempted execution (in the case of player-driven events), probability of successful execution, and execution time (the amount of time elapsed between executing an action and the point when the outcome of the execution is known). In consequence, there is an incentive for the actors to determine strategies (i.e., recipes telling the actors at which points in time or under what circumstances to perform particular actions) that result in achieving the main goal while optimizing some of the above parameters. In order to perform such analysis, the authors use *stochastic timed automata* (STA) [26]. Each of the nodes of the underlying diagram is translated into an STA which models attempted execution of a basic event (if the node represents a basic event) or the behavior of a gate in the presence of input (if the node is not a basic event). Similarly, a way of encoding strategies of the actors as STAs is presented. Special automaton ensures the correct order of propagation of the logical values through the automata and synchronization between them. The parallel composition of all the above automata, an STA itself, constitutes semantics of ADDs which corresponds to the possible realizations of the modeled attack–defense scenario under the given pair of strategies.

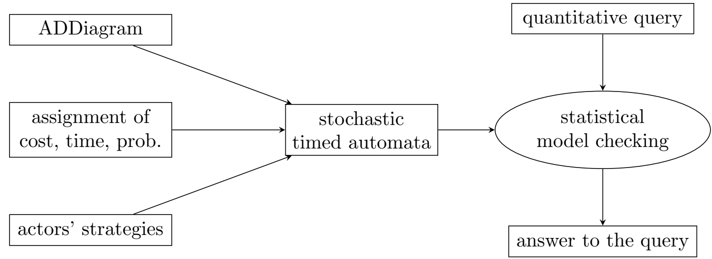


Fig. 17. STA-based analysis of ADDs by [51]

Analysis of STAs is known to be computationally difficult. In particular, determining a strategy optimal wrt the probability of reaching demanded state is in general case undecidable [28]. Hence, in order to address questions such as "what is the probability of success under given strategy?" or "what is the expected cost of a successful attack under given strategy?" Hermanns et al. employ tools for statistical model checking, namely the Modest tool-set [46, 49]. The values of interest are obtained by running a number of simulations and analyzing the resulting traces.

7 PROBABILISTIC ANALYSIS

The standard bottom-up algorithm sketched by Schneier in [101] can be applied to compute the probability of a successful attack scenario modeled with an attack tree. To do so, one needs to assign probability values to the leaves of the tree and then propagate them recursively to the parent nodes, according to the following formulæ: for a node having k children with respective probabilities p_i, one gets $1 - \Pi_{i=1}^{k}(1 - p_i)$ if the node is disjunctive, and $\Pi_{i=1}^{k} p_i$ if the node is conjunctive. However, such a computation has two main drawbacks. First, it requires the user to provide the input probability values for all basic actions. Such *data are hardly available* and thus often replaced with historical frequency estimations, i.e., how many times the action was successfully executed over a given period of time, in the past. However, frequency and probability are not the same measure: in this

context, frequency is about the past and probability is about the future. Second, a quick look at the above formulæ suffices to notice that such bottom-up computation procedure assumes that *all basic actions are independent*. This assumption, however, is rarely reflected in real-life situations, where most of the basic actions composing the attacks are strongly dependent on each other.

In this section, we present how the probabilistic security analysis with attack trees has evolved since the proposal of Schneier and how formal methods can improve it from the usability point of view. We start, in Section 7.1, with a method computing a probability distribution for an attack tree instead of a single probability value. In Section 7.2, we describe how to augment ADTrees with Bayesian networks to capture the dependencies between the basic actions involved in an attack-defense scenario. Stochastic games are used in Section 7.3 to perform probabilistic analysis of ADTrees, and finally, Section 7.4 uses probabilistic model checking to analyze attack trees.

7.1 Propagation of probability distribution on attack trees

The work of Arnold et al. presented in [8] has been motivated by the observation that the probability of an attack being successful increases with the time available to the attacker. In other words, "when given enough time, any system can be compromised". Inspired by this remark, the authors of [8] develop a framework for *time-dependent probabilistic* analysis of attack trees. Instead of simple probability values, *probability distributions* (expressing the probability as a function of time) are assigned to the leaves of an attack tree, and are propagated up to its root node. Thanks to this approach, one can estimate how much time is necessary so that the attacker succeeds with a given probability, but also check whether within a given time the probability of success is not greater than a critical threshold.

Attack trees considered in this work are composed of basic attack steps (BAS) represented by the leaves, and the refined nodes labeled by one of the three possible gates: `OR`, `AND`, and `SEQ`. The meaning of the `OR` and `AND` gates is standard. The sequential `SEQ` gate models that some basic steps can only be executed after some other steps have been successfully completed. To represent the distribution of its execution time, a cumulative distribution function (CDF) is assigned to every BAS. Arnold et al. discuss that these CDFs can be obtained in two ways: either historical or empirical data are provided to a fitting tool, such as G-Fit [104], which produces a matching CDF, or an expert estimates the mean time t necessary to execute a BAS and the exponential distribution $\exp(1/t)$ is employed. The use of the exponential distribution is justified by the fact that amongst all distributions with a given mean, it is the one that has maximal entropy. To obtain the CDF for the refined nodes, the distributions of the child nodes are aggregated using the operations of minimum (for `OR` gates), maximum (for `AND` gates), and convolution (for `SEQ` gates) of CDFs.

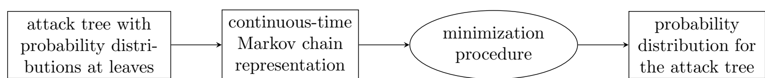


Fig. 18. Probability distribution on attack trees by [8]

To gain on efficiency, specific distributions, called *acyclic phase-type distributions* (APH), are used. Indeed, any continuous distribution function can be approximated arbitrarily closely by an APH distribution [59]. In addition, APHs are closed under minimum, maximum, and convolution. Phase-type distributions are probability distributions of the time needed to reach a final state in specific *continuous-time Markov chains* (CTMC). However, such a CTMC representation of an APH is not unique and the size of different representations may vary substantially. Finding the smallest representation is necessary because the application of minimum, maximum, and convolution operators may yield an exponential blow-up in the CTMC

representation. Arnold et al. refer to existing polynomial-time algorithms that effectively compress the size of APH representations [94] and that they have implemented in a prototype tool. The approach is illustrated in Fig. 18.

To position this work in the context of previously existing methods, the authors of [8] show that, in the case of classical attack trees, i.e., without SEQ gates, their time-dependent probabilistic analysis is a conservative extension of the standard bottom-up evaluation for probability. In other words, the time-dependent analysis may be seen as the static analysis for each point of time. This means that the probability of success by time t (in the time-dependent analysis) is equal to the probability that after time t the attack will be successful (computed in a standard bottom-up way). Together with [88] (for attack modeling) and [58] (for attack–defense modeling), the work of Arnold et al. is one of the rare approaches using probability distributions rather than probability points. One of its notable characteristics is that, by suggesting the use of exponential distributions, it offers a way to overcome the problem of providing input probability values for the leaves of attack trees.

7.2 Combining Bayesian networks and attack–defense trees

The objective of the approach presented by Kordy et al. in [71] is to lift the independence assumption between basic actions underlying the original bottom-up evaluation, and to compute the probability of a successful attack while taking dependencies between basic actions into account. The framework has been developed for ADTrees [68], hence it also applies to classical attack trees without explicit countermeasure nodes. The authors of [71] propose to complement an ADTree with a *Bayesian network* representing stochastic dependencies between basic actions present in the tree. The two graphs – the ADTree and the accompanying Bayesian network – are then translated into a semiring valuation, and a well-known algorithm called *fusion* is applied to compute the probability of interest. The framework, illustrated in Fig. 19, has first been presented in [70] and its extended version has been published in [71].

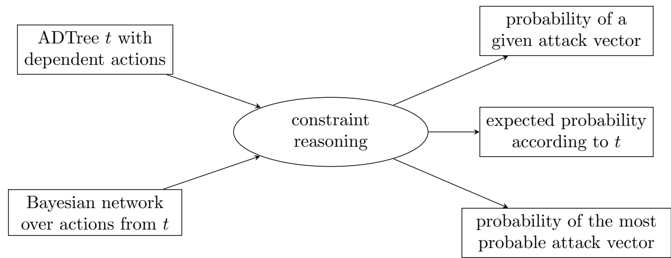


Fig. 19. Probability on ADTrees with dependencies by [71]

A Bayesian network [86] is a directed acyclic graph whose vertices correspond to variables with finite domains and where edges represent which variables are dependent on each other. The goal of a Bayesian network is to graphically depict a joint probability distribution over such a finite set of variables. To do so, each vertex of a Bayesian network is equipped with a conditional probability table expressing what is the probability of a vertex conditioned on its predecessor vertices. The joint probability distribution p of a Bayesian network over n variables $X_1, \ldots, X_n$ is given by $p(X_1, \ldots, X_n) = \Pi_{i=1}^{n} p(X_i \mid \mathrm{pred}(X_i))$, where $\mathrm{pred}(X_i)$ denotes the predecessors of X_i in the network.

In [71], an ADTree t is formally modeled as Boolean function f_t whose set of variables, denoted as var_t, corresponds to basic actions of the tree. In this representation, basic actions are thus independent. To capture possible stochastic dependencies between these actions, a Bayesian network, denoted BN_t, is constructed. The vertices of the Bayesian network BN_t correspond to basic actions of the ADTree t, i.e., to the variables of the Boolean function f_t. The objective of BN_t is to represent dependencies that are *not captured* by the structure of the ADTree. Therefore, the edges of BN_t

complement the refinement information present in t, and they need to be manually added by the expert modeling the attack–defense scenario.

By multiplying[5] Boolean function f_t with the joint probability distribution p_t of the Bayesian network BN_t one can express three main probability problems. Let $\mathbf{x} \in \{0,1\}^{\mathrm{var}_t}$ denote a vector representing which actions from var_t are attempted and which are not.

- If $\mathbf{x}$ represents an attack vector wrt t, i.e., an assignment satisfying f_t, then the probability of this vector being successful is expressed as $f_t(\mathbf{x}) \times p_t(\mathbf{x})$.
- The probability of attacking successfully according to ADTree t is expressed as $P(t) = \sum_{\mathbf{x} \in \{0,1\}^{\mathrm{var}_t}} f_t(\mathbf{x}) \times p_t(\mathbf{x})$.
- The success probability of the most probable attack vector wrt t is given by $P_{\max}(t) = \max_{\mathbf{x} \in \{0,1\}^{\mathrm{var}_t}} f_t(\mathbf{x}) \times p_t(\mathbf{x})$.

One can notice that the computation of $P(t)$ and $P_{\max}(t)$ grows exponentially with the number of basic actions in t. Thus, the direct computation using the above formulæ is not possible in the case of large, real-life ADTrees. Kordy et al. address this issue by using methods from the domain of *constraint reasoning*. The objective is to represent the formulæ for $P(t)$ and $P_{\max}(t)$ in a factorized form and make use of standard algorithms that exploit such a factorized form to compute $P(t)$ and $P_{\max}(t)$ in an efficient way. It turns out that this task is equivalent to solving classical *inference problems* over specific semirings. These problems can be solved using existing local computation algorithm known as *fusion* or *variable elimination*. Its complexity is bounded by a structural parameter, called *tree width* [98], and does not necessarily depend on the number of the variables in the problem. The approach presented in [71] has been automatically tested with the help of an open-source tool Nenok [91] providing a library of local computation algorithms, including fusion.

Finally, the authors of [71] showed that the probability computations using their approach are compatible with the propositional semantics for ADTrees, in the sense defined in [68]. This means that the results obtained on propositionally equivalent ADTrees are equal. Also, if the ADTree does not contain any dependent actions, the computation of $P(t)$ coincides with the standard bottom-up computation. A particularity of the approach of [71] is that the security model of ADTree and the dependency model of Bayesian network are kept separated. The ADTree represents refinements and countermeasures, while the Bayesian network captures additional dependencies between basic actions that are not covered by the tree. This is in contrast with other existing approaches, like for instance the one presented in [44], where the Bayesian network replaces an attack tree by covering all its nodes (not only the basic actions) and refinements which may result in Bayesian networks of large size. The bottle neck of the framework from [71] is the necessity of providing conditional probability tables for all basic actions involved in the considered ADTree.

7.3 Stochastic game interpretation of attack–defense trees

To overcome the limitations of usual static analysis of scenarios modeled with ADTrees, Aslanyan et al. propose a more dynamic approach in [13]. The formalism of ADTrees is extended with sequential conjunctive and sequential disjunctive nodes, to capture temporal or causal dependencies between the goals of the actors. With the basic actions being given an assignment of cost of attempted execution and probability of successful execution, the aim is to synthesize strategies for the actors that satisfy given constraints on the two parameters. Intuitively, a strategy provides an actor with information on what actions to perform, as well as in which order or under which circumstances particular actions should be executed. Formally, the strategies are represented as *decision trees*.

[5] To make this multiplication possible, the logical operators are expressed using algebraic operations: max for $\vee$ and $\times$ for $\wedge$.

They are derived from a specific *stochastic two-player game* (STG) [83] that the underlying ADTree is transformed into.

In order to analyze an ADTree, taking the order in which actions are executed into account, the authors of [13] propose a way of transforming the ADTree into an STG, see Fig. 20. To explicitly reason about strategies available to the players in the stochastic game, they use probabilistic model checking techniques for stochastic games based on the *probabilistic alternating-time temporal logic with rewards* (rPATL) [31]. This allows for expressing and answering questions such as "can the defender ensure that the probability of a successful attack is less than a given threshold?" or "what strategy of the attacker maximizes the probability of a successful attack?". An extension of rPATL [32] is employed to synthesize memoryless strategies (or verify their existence) satisfying given constraints on both parameters under consideration, i.e., a bound on the probability of a successful attack and a bound on the expected cost of implementing a strategy by one of the actors. The actual analysis of the game is performed by the PRISM-GAMES tool [77]. Apart from answering the above-mentioned questions, the tool can also present the Pareto optimal strategies (see Section 5.1; note however, that here the *expected*, and not the *exact*, cost is considered).

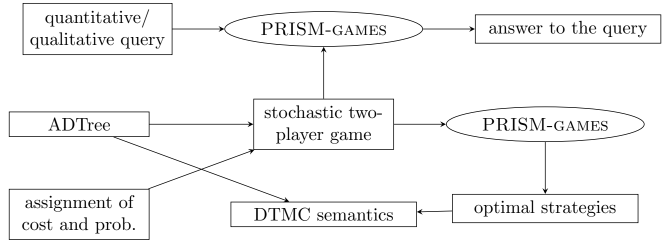


Fig. 20. PRISM-GAMES for ADTrees by [13]

Strategies of the actors in an ADTree are intuitively represented using a variant of decision trees. Given a pair of strategies to be implemented by the actors, the possible realizations of the modeled scenario are represented as a *discrete-time Markov chain* (DTMC) [92]. The equivalence between those strategies and the ones originating from the corresponding STG, as well as ways of obtaining the former given the latter, is presented. Finally, Aslanyan et al. implement a prototype tool that translates an ADTree into a specification of the corresponding STG that is accepted as input by the PRISM-GAMES tool.

The presented framework is developed under the assumption that the sequential nodes present in an ADTree cannot have non-sequential nodes among their ancestors. For the rest of this section let us refer to a maximal subtree of an ADTree that does not contain sequential nodes as simply *subtree*. We observe that the authors of [13] do not explicitly state the way in which they interpret multiple occurrences of a single basic action in an ADTree. However, one can deduce from the procedure constructing an STG that multiple nodes labeled with the same basic action and belonging to the same subtree are interpreted as the same single instance of the action. On the contrary, multiple occurrences originating from different subtrees are interpreted as distinct instances of the action.

7.4 Probabilistic model checking for attack trees

In [12], Aslanyan and Nielson develop a framework for quantitative analysis of AND/OR attack trees, using probabilistic model checking of *Markov decision processes* (MDPs) *with reward structure* [95]. They introduce a logic for expressing quantitative properties that combine the probability of success with the exact cost of attacks, and they develop a model checking algorithm for their logic. An attack tree is translated into a Markov decision process, and the verification of the aforementioned quantitative properties is reduced to the model checking of MDPs with reward structure, as summarized in Fig. 21.

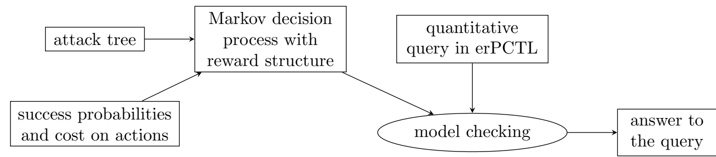


Fig. 21. Probabilistic model checking of attack trees by [12]

A Markov decision process is an operational state-transition model that combines non-deterministic and probabilistic transitions. A reward structure adds a reward (a positive integer) to each transition. In the Markov decision process resulting from a translation of an attack tree, non–deterministic transitions represent the choice of the attacker between performing or not an action, and the probabilistic transitions reflect the success or failure of the realization of actions. The choices of the attacker are made in the initially chosen order over basic actions. In this work, basic actions are assumed to be independent, so an arbitrary order can be fixed on the set of basic actions.

The logic introduced in the paper, namely the *Probabilistic Computational Tree Logic with Exact Rewards* (erPCTL), is an extension of *Probabilistic Computational Tree Logic* (PCTL) [48] which is a standard logic for MDPs, and the *Probabilistic Computational Tree Logic with Rewards* (rPCTL) [6], which itself is an extension of PCTL. In contrast to the exact cost (which is independent from probabilities), rPCTL considers only the expected cost of a run, i.e., the cost of a transition is multiplied by the probability of firing this transition. The authors of [12] introduce the erPCTL logic which adds two additional operators related to *exact costs* to rPCTL. The first operator asks about the probability of success of attacks satisfying a given path formula whose cost is in a given interval, and the second operator asks whether the costs of all attacks satisfying a given path formula are in a given interval. Finally, the model checking algorithm for erPCTL is proposed. This algorithm is polynomial in the size of the MDP, but the translation of an attack tree to an MDP introduces an exponential blowup.

8 WHERE TO TAKE IT FROM HERE

Since the publication of the first result on the formalization of attack trees [81], a lot of fundamental research on the topic has been performed. This effort led to numerous publications in renowned venues in the field of foundations for security (CSF, ESORICS, POST), formal methods (FORMATS, FASE), and logic (*Journal of Logic and Computation*, *Fundamenta Informaticæ*). The number of recently published articles shows that, during the last decade, graphical security modeling became a stand-alone research area, with several Ph.D. theses on the topic [9, 14, 74, 79, 96, 102], numerous national and international research projects, and a dedicated dissemination event – the GraMSec workshop (http://gramsec.uni.lu/).

In this survey, we focused only on foundational approaches providing mathematical background for exploiting formal methods to support the attack tree-based modeling. However, more research on attack trees in general has recently been performed. For instance, the work on attack tree engineering in Isabelle allows one to automate the handling of attack trees [61, 62]. Software development and testing based on security patterns has been supported with attack–defense trees in [96, 97]. The usage of attack trees within the red teaming activities has been proposed in [19]. Practical case studies have also been performed [36, 37, 40]. Numerous other directions of using attack tree-based modeling and analysis have been investigated, but they fall outside the scope of this paper.

The next biggest challenge for the scientific community working in the field will be to make sure that *industry practitioners can take the full advantage of the formal solutions presented in this article*. A promising initiative in this direction has been taken by Kumar et al. who created the

Table 4. Quantitative approaches and example queries (AT stands for attack tree, ADT for attack–defense tree)

Sec.	Model	Formalism	Example queries	Supporting tool
5.1	ADT	Bottom-up, Pareto efficiency	*What is the attack probability?* *What is the maximum probability and the minimum cost of an attack?*	ATE
5.2	ADT	Integer linear programming	*What is an optimal set of countermeasures for a given budget?* *Which set of countermeasures maximizes the minimal investment of the attacker?*	Prototype
5.3	AT	Bottom-up, Weight reduction	*Is the cost of a cheapest attack greater than a given threshold?*	–
5.4	ADT	Bottom-up	*What is the minimal cost/time/difficulty of an attack?*	–
6.1	AT	Priced timed automata	*What is the minimal time/resources/skill level needed for a successful attack?* *What are the top-10 worst attacks?*	UPPAAL CORA ATTOP
6.2	ADT	Stochastic timed automata	*What is the probability that an attack succeeds within a given time?* *What is the expected cost of the attacker within a given time*	UPPAAL
6.3	ADD	Stochastic timed automata	*What is the probability of success under a given strategy?* *What is the expected cost of a successful attack under a given strategy?*	MODEST
7.1	AT	Continuous-time Markov chains	*What is the attack probability distribution over time?*	Prototype
7.2	ADT	Bayesian network	*What is the probability of a given attack vector?* *What is the probability of the most probable attack vector?* *What is the expected probability of attacking according to an ADTree?*	NENOK
7.3	ADT	Stochastic games, discrete-time Markov chains	*Can the defender ensure that the probability of a successful attack is less than a given threshold?* *What strategy of the attacker maximizes the probability of a successful attack?*	PRISM-GAMES
7.4	AT	Markov decision processes	*What is the minimum cost of a successful attack?* *Is there a way to attack the system within the available budget?* *Is the cost of all successful attacks within a given budget?* *Is the cost of all successful attacks greater than a given threshold?* *What is the maximum probability of an attack with cost at most x?*	–

ATTop platform [76]. ATTop acts as a bridge between the Uppaal model checker and existing attack tree tools, in particular Attack Tree Evaluator (ATE) [10], ATCalc [7], and ADTool [41]. It allows the attack tree users to benefit from the analysis methods relying on timed automata, like those described in Section 6, without necessarily being proficient in this formalism. Extending ATTop to static quantitative approaches presented in Section 5 and stochastic solutions of Section 7 would be of great value for industrial users.

A platform like ATTop is helpful in analyzing attack trees, but it still requires a security expert to input the attack tree to be analyzed. The second major challenge is thus concerned with automated attack tree generation. As presented in Section 4, this direction has been extensively explored, and prototypes like ATSyRA (for physical context) or the tool developed by the TREsPASS consortium (for socio-technical context) exist. These tools have already proven that automating the generation of security models is possible and produces exploitable trees. However, *optimization work* is still necessary before such tools can handle large-scale systems. Automating the selection of countermeasures, and thus going from attack trees to ADTrees, could also be envisioned. Ideally, a fully-fledged tool chain should be able to generate attack trees from a domain-specific description of a system or requirements, analyze possible attack vectors to identify the most vulnerable elements of the system, and propose countermeasures to improve its security sufficiently.

An issue closely related to the automated generation of attack trees is the *format and the meaning of the node labels*. On the one hand, the labels should be short so that visual aspect of attack trees is not lost. On the other hand, labels that are too laconic may be a cause of a miscomprehension leading to falsified analysis results. Identifying syntactically different labels that express the same goals is necessary to make the process of attack tree creation composable. Composability is important when trees are built semi-automatically by reusing attack tree libraries or previously constructed models. A possible solution could be to apply text mining techniques to the documents describing existing attack trees or their templates, in order to devise clear and unique labels.

From the semantical perspective, extending original attack trees with SAND (to model causality and temporal relations) as well as formalizing the meaning of the repeated nodes is a good starting point to deal with dependencies between actions or goals represented by the node labels. However, *other than causal or temporal dependencies* may also be present in a security scenario. For instance, an administrator of a Linux machine can either disable the root account or create a strong password, but both measures cannot be used simultaneously. Thus, formalizing the semantics of a refinement expressing conflicting or mutually exclusive options could be a useful addition to attack trees. Furthermore, classical ADTrees contain a unique construct to denote countermeasures, but in practice, preventive (e.g., security training) and reactive (e.g., blocking a credit card after its theft) measures are not alike. A prevention makes the attack it protects against unfeasible, whereas a reaction allows an attacker to act but disables the attack's consequences. Consequently, depending on the type of the countermeasure applied, the attacker will or will not execute some of their actions. This is crucial for the security analysis, because it may impact the attack's cost or probability. We are currently working on developing formal foundations of ADTrees with preventions and reactions to augment the expressive power of ADTrees and make them more appropriate for the analysis of real-life security problems.

Regarding quantitative analysis of security, very few researchers have investigated the problem of *sensitivity analysis* in the context of attack trees, namely, how even a minor modification of the input values influences the output of the computation. Since getting the input values for attack tree computations is a well-known practical problem, estimating the accuracy of the computation result depending on the input error or uncertainty would be of great help to the users. The objective would be to assess how precise the inputs need to be so that the analysis is not meaningless. Of

course, due to the size of attack trees in practice, such sensitivity analysis cannot be performed manually by simply playing with possible input values. An interesting, from the mathematical perspective, research direction would be to explore standard variance or regression-based methods or alternative approaches based on emulators to address this problem.

REFERENCES

[1] 2005. Uppaal Cora. (2005). Retrieved May 29, 2018 from http://people.cs.aau.dk/~adavid/cora/
[2] 2014. ATSyRA. (2014). Retrieved May 29, 2018 from https://gforge.inria.fr/plugins/mediawiki/wiki/building/index.php/
[3] 2018. ATSyRA Studio. (2018). Retrieved Nov 16, 2018 from http://atsyra2.irisa.fr/
[4] Rajeev Alur, Mikhail Bernadsky, and P. Madhusudan. 2004. Optimal Reachability for Weighted Timed Games. In *ICALP (LNCS)*, Vol. 3142. Springer, 122–133.
[5] Rajeev Alur and David Dill. 1990. Automata for modeling real-time systems. In *ICALP (LNCS)*, Vol. 443. Springer, 322–335.
[6] Suzana Andova, Holger Hermanns, and Joost-Pieter Katoen. 2004. Discrete-time rewards model-checked. In *FORMATS (LNCS)*, Vol. 2791. Springer, 88–104.
[7] Florian Arnold, Axel Belinfante, Freark van der Berg, Dennis Guck, and Mariëlle Stoelinga. 2013. DFTCalc: A Tool for Efficient Fault Tree Analysis. In *SAFECOMP (LNCS)*, Vol. 8153. Springer, 293–301.
[8] Florian Arnold, Holger Hermanns, Reza Pulungan, and Mariëlle Stoelinga. 2014. Time-Dependent Analysis of Attacks. In *POST (LNCS)*, Vol. 8414. Springer, 285–305.
[9] Zaruhi Aslanyan. 2016. *Stochastic Model Checking of Socio-Technical Models.* Ph.D. Dissertation. Technical University of Denmark, Denmark.
[10] Zaruhi Aslanyan. 2016. TREsPASS toolbox: Attack Tree Evaluator. (2016). Retrieved May 29, 2018 from https://vimeo.com/145070436 presentation of a tool developed for the EU project TREsPASS.
[11] Zaruhi Aslanyan and Flemming Nielson. 2015. Pareto Efficient Solutions of Attack–Defence Trees. In *POST (LNCS)*, Vol. 9036. Springer, 95–114.
[12] Zaruhi Aslanyan and Flemming Nielson. 2017. Model checking exact cost for attack scenarios. In *POST (LNCS)*, Vol. 10204. Springer, 210–231.
[13] Zaruhi Aslanyan, Flemming Nielson, and David Parker. 2016. Quantitative verification and synthesis of attack–defence scenarios. In *CSF*. IEEE Computer Society, 105–119.
[14] Maxime Audinot. 2018. *Assisted design and analysis of attack trees.* Ph.D. Dissertation. University Rennes 1, France.
[15] Maxime Audinot, Sophie Pinchinat, and Barbara Kordy. 2017. Is My Attack Tree Correct?. In *ESORICS (LNCS)*, Vol. 10492. Springer, 83–102.
[16] Maxime Audinot, Sophie Pinchinat, and Barbara Kordy. 2018. Guided design of attack trees: a system-based approach. In *CSF*. IEEE Computer Society, 61–75.
[17] Maxime Audinot, Sophie Pinchinat, François Schwarzentruber, and Florence Wacheux. 2018. Deciding the Non-emptiness of Attack Trees. In *GraMSec 2018 (LNCS)*, Vol. 11086. Springer, 13–30.
[18] Alessandra Bagnato, Barbara Kordy, Per Håkon Meland, and Patrick Schweitzer. 2012. Attribute Decoration of Attack–Defense Trees. *IJSSE* 3, 2 (2012), 1–35.
[19] Matteo Beccaro. 2018. Attack Trees Methodology and Application in Red Teaming Operations. (2018). https://conference.hitb.org/hitbsecconf2018pek/materials/D1T1%20-%20Attack%20Trees%20-%20Methodology%20and%20Application%20in%20Red%20Teaming%20Operations%20-%20Matteo%20Beccaro.pdf D-HITBSecConf.
[20] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. 2004. *A Tutorial on Uppaal.* LNCS, Vol. 3185. Springer, 200–236.
[21] Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. 2004. Priced Timed Automata: Algorithms and Applications. In *FMCO (LNCS)*, Vol. 3657. Springer, 162–182.
[22] Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. 2005. Optimal Scheduling Using Priced Timed Automata. *SIGMETRICS Perform. Eval. Rev.* 32, 4 (March 2005), 34–40.
[23] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. 2005. lp_solve: Open source (Mixed-Integer) Linear Programming system. (2005). Retrieved June 10, 2018 from http://lpsolve.sourceforge.net/5.5/ Version 5.5.2.5, dated September 24, 2016.
[24] Dimitris Bertsimas and John Tsitsiklis. 1997. *Introduction to Linear Optimization.* Athena Scientific.
[25] Stefano Bistarelli, Fabio Fioravanti, Pamela Peretti, and Francesco Santini. 2012. Evaluation of complex security scenarios using defense trees and economic indexes. *J. Exp. Theor. Artif. Intell.* 24, 2 (2012), 161–192.
[26] Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. 2006. MODEST: A Compositional Modeling Formalism for Hard and Softly Timed Systems. *IEEE Trans. Software Eng.* 32, 10 (2006), 812–830.

[27] Angèle Bossuat and Barbara Kordy. 2018. Evil Twins: Handling Repetitions in Attack–Defense Trees – A Survival Guide. In *GraMSec 2017 (LNCS)*, Vol. 10744. Springer, 17–37.
[28] Patricia Bouyer and Vojtech Forejt. 2009. Reachability in Stochastic Timed Games. In *ICALP (2) (LNCS)*, Vol. 5556. Springer, 103–114.
[29] Thomas Brihaye, Véronique Bruyère, and Jean-François Raskin. 2004. Model-Checking for Weighted Timed Automata. In *FORMATS/FTRTFT (LNCS)*, Vol. 3253. Springer, 277–292.
[30] Ahto Buldas, Aleksandr Lenin, Jan Willemson, and Anton Charnamord. 2017. Simple Infeasibility Certificates for Attack Trees. In *IWSEC (LNCS)*, Vol. 10418. Springer, 39–55.
[31] Taolue Chen, Vojtech Forejt, Marta Z. Kwiatkowska, David Parker, and Aistis Simaitis. 2013. Automatic verification of competitive stochastic systems. *Formal Methods in System Design* 43, 1 (2013), 61–92.
[32] Taolue Chen, Vojtech Forejt, Marta Z. Kwiatkowska, Aistis Simaitis, and Clemens Wiltsche. 2013. On Stochastic Games with Multiple Objectives. In *MFCS (LNCS)*, Vol. 8087. Springer, 266–277.
[33] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All About Maude – A High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic.* Springer.
[34] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.
[35] EAC Advisory Board and Standards Board. 2009. Election Operations Assessment – Threat Trees and Matrices and Threat Instance Risk Analyzer (TIRA). (2009). Retrieved June 13, 2018 from https://www.eac.gov/assets/1/28/Election_Operations_Assessment_Threat_Trees_and_Matrices_and_Threat_Instance_Risk_Analyzer_(TIRA).pdf
[36] Barbara Fila and Wojciech Wideł. 2019. Attack–defense trees for abusing optical power meters: A case study and the OSEAD tool experience report. (2019). (To appear in GraMSec'19).
[37] Marlon Fraile, Margaret Ford, Olga Gadyatskaya, Rajesh Kumar, Mariëlle Stoelinga, and Rolando Trujillo-Rasua. 2016. Using Attack–Defense Trees to Analyze Threats and Countermeasures in an ATM: A Case Study. In *PoEM (LNBIP)*, Vol. 267. Springer, 326–334.
[38] Olga Gadyatskaya. 2015. How to Generate Security Cameras: Towards Defence Generation for Socio-Technical Systems. In *GraMSec 2015 (LNCS)*, Vol. 9390. Springer, 50–65.
[39] Olga Gadyatskaya, René Rydhof Hansen, Kim Guldstrand Larsen, Axel Legay, Mads Chr. Olesen, and Danny Bøgsted Poulsen. 2016. Modelling Attack–defense Trees Using Timed Automata. In *FORMATS (LNCS)*, Vol. 9884. Springer, 35–50.
[40] Olga Gadyatskaya, Carlo Harpes, Sjouke Mauw, Cédric Muller, and Steve Muller. 2016. Bridging Two Worlds: Reconciling Practical Risk Assessment Methodologies with Theory of Attack Trees. In *GraMSec 2016 (LNCS)*, Vol. 9987. Springer, 80–93.
[41] Olga Gadyatskaya, Ravi Jhawar, Piotr Kordy, Karim Lounis, Sjouke Mauw, and Rolando Trujillo-Rasua. 2016. Attack Trees for Practical Security Assessment: Ranking of Attack Scenarios with ADTool 2.0. In *QEST (LNCS)*, Vol. 9826. Springer, 159–162.
[42] Olga Gadyatskaya, Ravi Jhawar, Sjouke Mauw, Rolando Trujillo-Rasua, and Tim A. C. Willemse. 2017. Refinement-Aware Generation of Attack Trees. In *STM (LNCS)*, Vol. 10547. Springer, 164–179.
[43] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102.
[44] Marco Gribaudo, Mauro Iacono, and Stefano Marrone. 2015. Exploiting Bayesian Networks for the Analysis of Combined Attack Trees. *Electr. Notes Theor. Comput. Sci.* 310 (2015), 91–111.
[45] David F. Haasl, Norman H. Roberts, William E. Veselay, and Francine F. Goldberg. 1981. *Fault Tree Handbook.* Technical Report. Systems and Reliability Research, Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Comission.
[46] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. 2013. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design* 43, 2 (2013), 191–232.
[47] René Rydhof Hansen, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, and Danny Bøgsted Poulsen. 2018. Quantitative Evaluation of Attack Defense Trees Using Stochastic Timed Automata. In *GraMSec 2017 (LNCS)*, Vol. 10744. Springer, 75–90.
[48] Hans Hansson and Bengt Jonsson. 1994. A logic for reasoning about time and reliability. *Formal aspects of computing* 6, 5 (1994), 512–535.
[49] Arnd Hartmanns and Holger Hermanns. 2014. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *TACAS (LNCS)*, Vol. 8413. Springer, 593–598.
[50] Thomas Henzinger, Zohar Manna, and Amir Pnueli. 1992. Timed Transition Systems. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems) (LNCS)*, Vol. 600. Springer, 226–251.
[51] Holger Hermanns, Julia Krämer, Jan Krcál, and Mariëlle Stoelinga. 2016. The Value of Attack-Defence Diagrams. In *POST (LNCS)*, Vol. 9635. Springer, 163–185.

[52] Jin B. Hong, Dong Seong Kim, Chun-Jen Chung, and Dijiang Huang. 2017. A survey on the usability and practical applications of Graphical Security Models. *Computer Science Review* 26 (2017), 1–16.
[53] Ross Horne. 2015. The Consistency and Complexity of Multiplicative Additive System Virtual. *Sci. Ann. Comp. Sci.* 25, 2 (2015), 245–316.
[54] Ross Horne, Sjouke Mauw, and Alwen Tiu. 2017. Semantics for Specialising Attack Trees Based on Linear Logic. *Fundam. Inform.* 153, 1-2 (2017), 57–86.
[55] Marieta Georgieva Ivanova, Christian W. Probst, René Rydhof Hansen, and Florian Kammüller. 2015. Attack Tree Generation by Policy Invalidation. In *WISTP (LNCS)*, Vol. 9311. Springer, 249–259.
[56] Marieta Georgieva Ivanova, Christian W. Probst, René Rydhof Hansen, and Florian Kammüller. 2015. Transforming Graphical System Models to Graphical Attack Models. In *GraMSec 2015 (LNCS)*, Vol. 9390. Springer, 82–96.
[57] Ravi Jhawar, Barbara Kordy, Sjouke Mauw, Sasa Radomirovic, and Rolando Trujillo-Rasua. 2015. Attack Trees with Sequential Conjunction. In *SEC (IFIP AICT)*, Vol. 455. Springer, 339–353.
[58] Ravi Jhawar, Karim Lounis, and Sjouke Mauw. 2016. A Stochastic Framework for Quantitative Analysis of Attack–Defense Trees. In *STM (LNCS)*, Vol. 9871. Springer, 138–153.
[59] Mary A. Johnson and Michael R. Taaffe. 1988. The denseness of phase distributions. (1988). School of Industrial Engineering Research Memoranda 88-20, Purdue University.
[60] Aivo Jürgenson and Jan Willemson. 2008. Computing Exact Outcomes of Multi-parameter Attack Trees. In *OTM Conferences (2) (LNCS)*, Vol. 5332. Springer, 1036–1051.
[61] Florian Kammüller. 2017. A Proof Calculus for Attack Trees in Isabelle. In *DPM/CBT@ESORICS (LNCS)*, Vol. 10436. Springer, 3–18.
[62] Florian Kammüller. 2018. Attack Trees in Isabelle. In *ICICS (LNCS)*, Vol. 11149. Springer, 611–628.
[63] Florian Kammüller and Christian W. Probst. 2013. Invalidating Policies using Structural Information. In *IEEE Symposium on Security and Privacy Workshops*. IEEE Computer Society, 76–81.
[64] Florian Kammüller and Christian W. Probst. 2014. Combining Generated Data Models with Formal Invalidation for Insider Threat Analysis. In *IEEE Symposium on Security and Privacy Workshops*. IEEE Computer Society, 229–235.
[65] Joost-Pieter Katoen and Mariëlle Stoelinga. 2017. Boosting Fault Tree Analysis by Formal Methods. In *ModelEd, TestEd, TrustEd (LNCS)*, Vol. 10500. Springer, 368–389.
[66] Robert M Keller. 1976. Formal verification of parallel programs. *Commun. ACM* 19, 7 (1976), 371–384.
[67] Barbara Kordy, Piotr Kordy, and Yoann van den Boom. 2016. SPTool - Equivalence Checker for SAND Attack Trees. In *CRiSIS (LNCS)*, Vol. 10158. Springer, 105–113.
[68] Barbara Kordy, Sjouke Mauw, Sasa Radomirovic, and Patrick Schweitzer. 2014. Attack–defense trees. *J. Log. Comput.* 24, 1 (2014), 55–87.
[69] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. 2014. DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer Science Review* 13-14 (2014), 1–38.
[70] Barbara Kordy, Marc Pouly, and Patrick Schweitzer. 2014. A Probabilistic Framework for Security Scenarios with Dependent Actions. In *iFM (LNCS)*, Vol. 8739. Springer, 256–271.
[71] Barbara Kordy, Marc Pouly, and Patrick Schweitzer. 2016. Probabilistic reasoning with graphical security models. *Inf. Sci.* 342 (2016), 111–131.
[72] Barbara Kordy and Wojciech Wideł. 2017. How well can I secure my system?. In *iFM'17 (LNCS)*, Vol. 10510. Springer, 332–347.
[73] Barbara Kordy and Wojciech Wideł. 2018. On quantitative analysis of attack–defense trees with repeated labels. In *POST (LNCS)*, Vol. 10804. Springer, 325–346.
[74] Rajesh Kumar. 2018. *Truth or Dare: Quantitative security risk analysis via attack trees.* Ph.D. Dissertation. University of Twente, The Netherlands.
[75] Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. 2015. Quantitative Attack Tree Analysis via Priced Timed Automata. In *FORMATS (LNCS)*, Vol. 9268. Springer, 156–171.
[76] Rajesh Kumar, Stefano Schivo, Enno Ruijters, Buğra M. Yildiz, David Huistra, Jacco Brandt, Arend Rensink, and Mariëlle Stoelinga. 2018. Effective Analysis of Attack Trees: a Model-Driven Approach. In *FASE (LNCS)*, Alessandra Russo and Andy Andy Schürr (Eds.), Vol. 10802. Springer, 56–73.
[77] Marta Kwiatkowska, David Parker, and Clemens Wiltsche. 2016. *PRISM-Games 2.0: A Tool for Multi-objective Strategy Synthesis for Stochastic Games*. LNCS, Vol. 9636. Springer, 560–566.
[78] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a Nutshell. *STTT* 1, 1-2 (1997), 134–152.
[79] Aleksandr Lenin. 2015. *Reliable and Efficient Determination of the Likelihood of Rational Attacks.* Ph.D. Dissertation. Tallinn University of Technology, Estonia.
[80] Aleksandr Lenin, Jan Willemson, and Dyan Permata Sari. 2014. Attacker Profiling in Quantitative Security Assessment Based on Attack Trees. In *NordSec (LNCS)*, Vol. 8788. Springer, 199–212.

[81] Sjouke Mauw and Martijn Oostdijk. 2005. Foundations of Attack Trees. In *ICISC (LNCS)*, Vol. 3935. Springer, 186–198.

[82] National Electric Sector Cybersecurity Organization Resource (NESCOR). 2015. Analysis of Selected Electric Sector High Risk Failure Scenarios, Version 2.0. (2015). Retrieved June 13, 2018 from http://smartgrid.epri.com/doc/NESCOR%20Detailed%20Failure%20Scenarios%20v2.pdf

[83] Abraham Neyman and Sylvain Sorin. 2003. *Stochastic Games and Applications*. NATO Science Series ASIC, Vol. 570. Kluwer Academic Publishers.

[84] Peter Niebert, Stavros Tripakis, and Sergio Yovine. 2000. Minimum-time reachability for timed automata. In *IEEE Mediteranean Control Conference*. IEEE, 8.

[85] Hanne Riis Nielson, Flemming Nielson, and Roberto Vigo. 2012. A Calculus for Quality. In *FACS (LNCS)*, Vol. 7684. Springer, 188–204.

[86] Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

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

[88] Ludovic Piètre-Cambacédès and Marc Bouissou. 2010. Attack and Defense Modeling with BDMP. In *MMM-ACNS (LNCS)*, Vol. 6258. Springer, 86–101.

[89] Sophie Pinchinat, Mathieu Acher, and Didier Vojtisek. 2014. Towards Synthesis of Attack Trees for Supporting Computer-Aided Risk Analysis. In *SEFM Workshops (LNCS)*, Vol. 8938. Springer, 363–375.

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

[91] Marc Pouly. 2010. NENOK – A Software Architecture for Generic Inference. *Int. J. on Artif. Intel. Tools* 19 (2010), 65–99.

[92] Nicolas Privault. 2013. Discrete-Time Markov Chains. In *Understanding Markov Chains: Examples and Applications*. Springer, 77–94.

[93] Christian W. Probst, Jan Willemson, and Wolter Pieters. 2015. The Attack Navigator. In *GraMSec 2015 (LNCS)*, Vol. 9390. Springer, 1–17.

[94] Reza Pulungan and Holger Hermanns. 2009. Acyclic Minimality by Construction—Almost. In *QEST*. IEEE Computer Society, 63–72.

[95] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[96] Loukmen Regainia. 2018. *Assisting in the development and testing of secure applications*. Ph.D. Dissertation. University Clermont Auvergne, France.

[97] Loukmen Regainia and Sébastien Salva. 2017. A Methodology of Security Pattern Classification and of Attack-Defense Tree Generation. In *ICISSP*. SciTePress, 136–146.

[98] N. Robertson and P.D. Seymour. 1983. Graph Minors I: Excluding a Forest. *J. Comb. Theory, Ser. B* 35, 1 (1983), 39–61.

[99] Arpan Roy, Dong Seong Kim, and Kishor S. Trivedi. 2012. Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees. *Security and Communication Networks* 5, 8 (2012), 929–943.

[100] Enno Ruijters and Mariëlle Stoelinga. 2015. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review* 15 (2015), 29–62.

[101] Bruce Schneier. 1999. Attack trees. *Dr. Dobb's journal* 24, 12 (1999), 21–29.

[102] Patrick Schweitzer. 2013. *Attack–Defense Trees*. Ph.D. Dissertation. University of Luxembourg, Luxembourg.

[103] Yann Thierry-Mieg. 2015. Symbolic Model-Checking Using ITS-Tools. In *TACAS (LNCS)*, Vol. 9035. Springer, 231–237.

[104] Axel Thümmler, Peter Buchholz, and Miklós Telek. 2006. A Novel Approach for Phase-Type Fitting with the EM Algorithm. *IEEE Trans. Dependable Sec. Comput.* 3, 3 (2006), 245–258.

[105] Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson. 2014. Automated Generation of Attack Trees. In *CSF*. IEEE Computer Society, 337–350.

[106] Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson. 2016. Discovering, quantifying, and displaying attacks. *Logical Methods in Computer Science* 12, 4 (2016).

[107] Jonathan D. Weiss. 1991. A system security engineering process. In *14th Annual NCSC/NIST National Computer Security Conference*. 572–581.

Received June 2018; revised December 2018; accepted May 2019