# Hackers vs. Security: Attack-Defence Trees as Asynchronous Multi-Agent Systems*

Jaime Arias[1], Carlos E. Budde[2], Wojciech Penczek[3], Laure Petrucci[1], Teofil Sidoruk[3,5], and Mariëlle Stoelinga[2,4]

[1] LIPN, CNRS UMR 7030, Université Sorbonne Paris Nord, Sorbonne Paris Cité, Villetaneuse, France
[2] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
[3] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
[4] Department of Software Science, Radboud University, Nijmegen, The Netherlands
[5] Warsaw University of Technology, Warsaw, Poland

**Abstract.** Attack-Defence Trees (ADTrees) are a well-suited formalism to assess possible attacks to systems and the efficiency of countermeasures. This paper extends the available ADTree constructs with reactive patterns that cover further security scenarios, and equips all constructs with attributes such as time and cost to allow for quantitative analyses. We model ADTrees as (an extension of) Asynchronous Multi-Agents Systems: EAMAS. The ADTree–EAMAS transformation allows us to quantify the impact of different agents configurations on metrics such as attack time. Using EAMAS also permits parametric verification: we derive constraints for property satisfaction, e.g. the maximum time a defence can take to block an attack. Our approach is exercised on several case studies using the Uppaal and IMITATOR tools. We developed the open-source tool adt2amas implementing our transformation.

## 1 Introduction

Over the past ten years of security analysis, multiple formalisms have been developed to study interactions between attacker and defender parties [28, 19, 22, 16, 26]. Among these, *Attack-Defence Trees* (ADTrees [22]) stand out as a graphical, straightforward formalism of great modelling versatility. However, research is thus far focused on bipartite graph characterisations, where nodes belong to either the attacker or defender party [23, 22, 8, 14]. This can model interactions between opposing players, but lacks expressiveness to analyse potential sources of parallelism when each party is itself formed of multiple agents.

*Agents distribution* over the tree nodes, i.e. which agent performs which task for which goal, can determine not only the performance but also the feasibility of an attack or defence strategy. For instance, a monitored double-locked gate may require two concurrent burglars, to steal goods before the alerted police arrives.

* This work was partially funded by the NWO project SEQUOIA, the PHC van Gogh project PAMPAS, the BQR project AMoJAS, and the IEA project PARTIES.

Likewise, distributed DoS attacks exploit multiplicity to overcome standard DoS countermeasures. Clearly, studying agents distribution *within the operations of an attack/defence party* is crucial to assess attacks and effective countermeasures. However and to our surprise, we find no literature studies focused in this topic.

To fill this gap we hereby model ADTrees in an agent-aware formalism, and study the mechanics of different agents distributions. Our approach permits quantifying performance metrics (e.g. cost and time) of attack/defence strategies under distinct agents coalitions. Employing modern verification tools—IMITATOR [4] and Uppaal [11]—we *reason about the impact of coalition strategies*, and *synthesise the value of the attributes that make them feasible*, such as the maximum time allowed for a defence mechanism to be triggered. In this way, we make an important step towards the analysis of more complex security scenarios.

**Contributions.** Concretely, in this paper we introduce: (***i***) a unified scheme for ADTree representation with counter- and sequential-operators, including a new construct to negate sequences; (***ii***) EAMAS: formal semantics to model ADTrees, where *all* nodes have attributes and can be operated by *agents*; (***iii***) compositional, sound and complete *pattern transformation rules* from ADTree to EAMAS, which can model ADTrees with shared subtrees; (***iv***) the open-source tool `adt2amas` [1] to translate ADTree models into EAMAS and generate IMITATOR models; (***v***) measurements of the impact of different *agents coalitions* on attack performance metrics, such as cost, exercised on several case studies; (***vi***) *synthesis of ADTree attributes* (e.g. time) that enable attack/defence strategies.

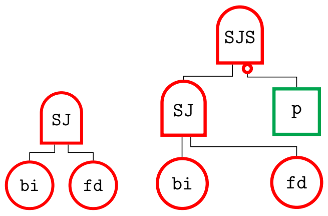
**Outline.** In Secs. 2 and 3 we review the basic notions of ADTrees and AMAS. Sec. 4 extends AMAS with attributes, to model ADTrees via the graph-based transformation patterns introduced in Sec. 5. The effectiveness of our approach is shown in Sec. 6, where we analyse three case studies from the literature, and demonstrate scalability. We conclude in Sec. 7 and discuss future research.

**Related work.** Attack-Defence Trees [22] extend Attack Trees with defensive counter-actions. Several analysis frameworks implement this formalism as Priced Timed Automata (PTA) [14], I/O-IMCs [7], Bayesian Networks (BN) [15], stochastic games [9], and so on—see surveys [23, 30]. Each framework computes queries for the underlying semantics: conditional probabilities for BNs, time of attacks/defences for PTAs, etc. In [25] a model driven approach is proposed to inter-operate across these frameworks. However, none of them analyses agent distributions *within* the attack/defence parties. Such studies are at the core of this work. Furthermore, most analyses operate on fully described models, where the attributes of all basic attack and defence nodes are known a priori. Instead we extend the work of [5] to ADTrees, *synthesising* (constraints for the) values of attributes that yield a successful attack or defence. Moreover, our EAMAS formalism offers a succinct representation amenable to state space reduction techniques [21]. This deploys lightweight analyses in comparison to other highly expressive formalisms, such as Attack-Defence Diagrams [16]. Via EAMAS we extend the work on Attack Trees in [18]: we give formal semantics to sequential order operators in ADTrees, that keep the order of events but abstract away their exact occurrence point in time, as usual in the literature [28, 10, 20, 23, 18, 25].

## 2 Attack-Defence Trees

### 2.1 The basic ADTree model

*Attack Trees* are graphical tree-like representations of attack scenarios. They allow for evaluating the security of complex systems to the desired degree of refinement. The *root* of the tree is the goal of the attacker, and the children of a node represent refinements of the node's goal into sub-goals. The tree *leaves* are (possibly quantified) basic attack actions. For instance, Fig. 1a shows a simplistic Attack Tree where the goal is to Steal Jewels from a museum (`SJ`), for which burglars must break in (node `bi`, an *attack leaf*) and force a display (`fd`). Nodes in the tree whose state depends on other nodes are called *gates*: `SJ` is an `AND` gate with two children.



(a) Attack Tree (b) ADTree

Fig. 1: Steal Jewels

*Attack-Defence Trees* [22] can model counter-actions of a defender: they represent an interplay between the actions of both attacker and defender. This can model mechanisms triggered by the occurrence of opposite actions. So for instance in the ADTree in Fig. 1b, the jewels burglary will succeed (`SJS`) only if all attack actions are performed, and the alerted police (node `p`, a *defence leaf*) does not stop them.

We define ADTree structures as shown in Table 1. The formal semantics of each construct will be later given in Sec. 5 in terms of (specialised) Multi-Agent Systems; here we simply give a natural language interpretation of such semantics. Since constructs are symmetric for attack and defence goals, Table 1 shows a comprehensive selection of structures. Here $D, d, d_1, \cdots, d_n \in \Sigma_d$ and $A, a, a_1, \cdots, a_n \in \Sigma_a$, where $\Sigma_d$ and $\Sigma_a$ are sets of defence and attack nodes, respectively. Graphically, triangular nodes stand for arbitrary subtrees that are children of a gate, and circular (resp. rectangular) nodes represent attack (resp. defence) leaves, i.e. basic actions that are no further refined. Table 1 thus reinterprets [22] using a unified gate notation along the lines of [25], including `CAND` gates that express counter-attacks or -defences, e.g. *counter defence* in the table.

Table 1 also introduces operators for a *choice* between a successful attack and a failing defence (named *no defence*), and vice-versa (*inhibiting attack*). These constructs, less usual in the literature [23], model realistic scenarios such as attack goals succeeding by security negligence rather than by performing costly attack. This is of interest for quantitative analyses of e.g. cost and probability.

Moreover, we consider sequential operators, which lack a standard interpretation for ADTrees. For Attack Trees, [18] proposes a sequentially ordered conjunction (`SAND`) where attacks succeed as soon as all children took place in the required order. This abstracts away the occurrence time point of events, and describes instead the *order* in which events must take place. Thus, `SAND` gates *enforce* sequential events and rule out parallel executions: this is a fundamental construct in multi-agent systems. For instance, Steal Jewels (`SJ`) in Fig. 1 is modelled with an `AND` gate. Let break-in (`bi`) take 10 min and force the display (`fd`) 5 min. If two attackers cooperate, an ADTree analysis could conclude that

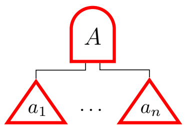
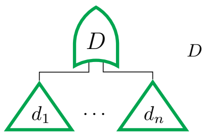
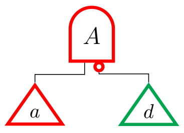
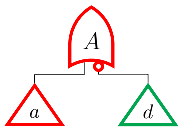
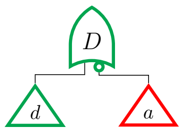
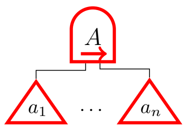
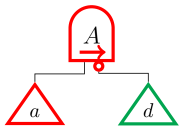
| Name | Graphics | Semantics |
|---|---|---|
| Attack | | $a \equiv$ "basic attack *action* $a$ done" |
| Defence | | $d \equiv$ "basic defence *action* $d$ done" |
| And attack |  | $A \equiv$ "attacks $a_1$ through $a_n$ done" |
| Or defence |  | $D \equiv$ "one of the defences $d_1$ through $d_n$ done" |
| Counter defence |  | $A \equiv$ "attack $a$ done and defence $d$ not done" |
| No defence |  | $A \equiv$ "either attack $a$ done or else defence $d$ not done" |
| Inhibiting attack |  | $D \equiv$ "either defence $d$ done or else attack $a$ not done" |
| Sequential and attack |  | $A \equiv$ "done attack $a_1$, then attack $a_2$, … then attack $a_n$" |
| Failed reactive defence |  | $A \equiv$ "done attack $a$ and then did not do defence $d$" |

Table 1: ADTree constructs (selection) and their informal semantics

attack `SJ` succeeds after 10 min. But `fd` depends logically on `bi`, since the display can only be forced after breaking in. Using instead a `SAND` gate for `SJ` enforces this sequentiality so that attacks cannot take less than 15 min. We integrate such `SAND` gates in our ADTree framework, as the *sequential and attack* in Table 1.

We further introduce `SCAND` gates: sequential gates that have attacks and defences as children. To the best of our knowledge, this is novel in a typed setting where subtrees (rather than leaves) can be assigned attack/defence goals. This contribution is conservative: it extends the `SAND` gates of [18] to coincide with previous works on sequential operators in defence-aware representations, e.g. Attack-Defence Diagrams [16]. We distinguish two scenarios: a successful attack followed by a failed counter defence (*failed reactive defence* in Table 1), and vice versa. We disregard the second scenario as uninteresting—it models defence goals which depend on attacks failing by themselves—and focus on the first one. `SCAND`s then model an attack goal that must overcome some counter defence, triggered only after the incoming attack has been detected.

## 2.2 Attributes and agents for ADTrees

Attributes (also "parameters" and "gains" [10, 8, 25]) are numeric properties of attack/defence nodes that allow for quantitative analyses. Typical attributes include cost and time: in the Steal Jewels example, the 10 min to break in is a time attribute of this attack leaf. In general, attributes are associated only with tree leaves, and used to compute e.g. the min/max time required by an attack.

**General attributes.** We extend attributes, from leaves, to all nodes in ADTrees, because a node need not be fully described by its children. An attribute is then given by a node's *intrinsic value*, and a *computation function*. For example, refine bi to be an AND gate between pick main lock (pml, 7 min) and saw padlock (sp, 2 min). Then it may take an extra minute to enter and locate the correct display, counted after pml and sp finished. In general, when the goal of a gate is successful, its attribute value is the result of its computation function applied to its intrinsic value and to the attributes of its children. For bi, if two burglars cooperate, the computation function is $\mathit{init\_time}(\texttt{bi})+\max(\mathit{init\_time}(\texttt{pml}), \mathit{init\_time}(\texttt{sp}))$. This allows for flexibility in describing different kinds of attributes, and gains special relevance when considering coalitions of agents, as we will further illustrate in Sec. 2.3. Moreover, attributes can be *parameters* as in [5]. We can synthesise constraints over parameters, such as $\mathit{init\_time}(\texttt{bi}) \leqslant 1$ min, e.g. to determine which attribute values make an attack successful.

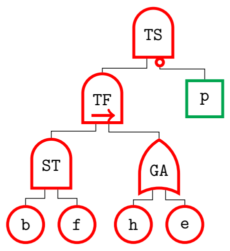
**Agents.** Each action described by an ADTree construct can be performed by a particular *agent*. Different attacks/defences could be handled by one or multiple agents, which allows to express properties on agents coalitions. For instance, in the Steal Jewels example of Fig. 1b, the minimal number of burglars required to minimise the SJS attack time is two: one to do bi and another to parallelly perform fd. If the SJ gate is changed to a SAND, then one burglar suffices, since bi and fd cannot be parallelised. Upon using the refinement $\texttt{bi} = \texttt{AND}(\texttt{pml}, \texttt{sp})$, then again a coalition of two burglars minimises the attack time, since pml and sp can be parallelised. Each node in the ADTree will thus be assigned to an agent, and a single agent can handle multiple nodes. In the general case, the only constraint is that no agent handles both attack and defence nodes. Notice that even modern formalisms for ADTrees such as [13] are oblivious of agents distributions: encoding them requires modifying the tree structure, e.g. changing an AND for a SAND to enforce the sequential occurrence of actions (i.e. they are carried out by the same agent). As we show in Sec. 4 and demonstrate empirically in Sec. 6, our semantics decouples the attack structure from the agent distribution. This permits to analyse and synthesise which distribution optimises a goal, e.g. achieve the fastest attack, without tampering with the ADTree model. Users can thus focus exclusively on the relevant studies of agent coalitions: this entails less error-prone and shorter computation times than in formalisms where agent distributions must be hacked into the ADTree structure.

**Conditional counter measures.** It may happen that a countering node has a successful or unsuccessful outcome depending on the attributes of its children. We therefore associate *conditions* with countering nodes, which are Boolean functions over the attributes of the ADTree. When present, the condition then comes as an additional constraint for the node operation to be successful.

### 2.3 Example: Treasure hunters

Our simple running example in Fig. 2 features thieves that try to steal a treasure in a museum. To achieve their goal, they first must access the treasure room, which involves bribing a guard (b), and forcing the secure door (f). Both actions are costly and take some time. Two coalitions are possible: either a single thief has to carry out both actions, or a second thief could be hired to parallelise b and f. After these actions succeed the attacker/s can steal the treasure (ST), which takes a little time for opening its display stand and putting it in a bag. If the two-thieves coalition is used, we encode in ST an extra €90 to hire the second thief—the computation function of the gate can handle this plurality—else ST incurs no extra cost. Then the thieves are ready to flee (TF), choosing an escape route to get away (GA): this can be a spectacular escape in a helicopter (h), or a mundane one via the emergency exit (e). The helicopter is expensive but fast while the emergency exit is slower but at no cost. Furthermore, the time to perform a successful escape could depend on the number of agents involved in the robbery. Again, this can be encoded via computation functions in gate GA.

As soon as the treasure room is penetrated (i.e. after b and f but before ST) an alarm goes off at the police station, so while the thieves flee the police hurries to intervene (p). The treasure is then successfully stolen iff the thieves have fled and the police failed to arrive or does so too late. This last possibility is captured by the condition associated with the treasure stolen gate (TS), which states that the arrival time of the police must be greater than the time for the thieves to steal the treasure and go away.



(a) ADTree

| **Name** | **Cost** | **Time** |
|---|---|---|
| TS (treasure stolen) | | |
| p (police) | €100 | 10 min |
| TF (thieves fleeing) | | |
| ST (steal treasure) | €$\{0, 90\}$ | 2 min |
| b (bribe gatekeeper) | €500 | 1 h |
| f (force arm. door) | €100 | 2 h |
| GA (get away) | | |
| h (helicopter) | €500 | 3 min |
| e (emergency exit) | | 10 min |

**Condition for TS:**
$\mathit{init\_time}(\mathtt{p}) > \mathit{init\_time}(\mathtt{ST}) + \mathit{time}(\mathtt{GA})$

(b) Attributes of nodes

Fig. 2: The treasure hunters

## 3 AMAS

*Asynchronous Multi-Agent Systems* (AMAS [17]) are a modern semantic model for the study of agents' strategies in asynchronous systems. They provide an analysis framework with efficient reachability checks even on non-trivial models. Technically, AMAS are similar to networks of automata that synchronise on shared actions, and interleave local transitions to execute asynchronously [12, 27, 17]. However, to deal with agents coalitions, automata semantics (e.g. for ADTrees) must resort to algorithms and additional attributes. In contrast, by linking protocols to agents, AMAS are a natural compositional formalism to analyse multi-agent systems.

**Definition 1 (Asynchronous Multi-Agent Systems [17]).** *An* asynchronous multi-agent system (AMAS) *consists of $n$* agents *$A = \{1, \ldots, n\}$, where each agent has an associated tuple $A_i = (L_i, \iota_i, Act_i, P_i, T_i)$ including (i) a set of* local states *$L_i = \{l_i^1, l_i^2, \ldots, l_i^{n_i}\}$; (ii) an* initial state *$\iota_i \in L_i$; (iii) a set of* actions *$Act_i = \{a_i^1, a_i^2, \ldots, a_i^{m_i}\}$; (iv) a* local protocol *$P_i : L_i \to 2^{Act_i}$ which selects the actions available at each local state; and (v) a (partial)* local transition function *$T_i \subseteq L_i \times Act_i \times L_i$ s.t. $(l_i, a, l_i') \in T_i$ for some $l_i' \in L_i$ iff $a \in P_i(l_i)$.*

Sets $Act_i$ need not be disjoint. $Act = \bigcup_{i \in A} Act_i$ and $Loc = \bigcup_{i \in A} L_i$ are resp. the set of all actions and all local states. For each action $a \in Act$, set $Agent(a) =$

$\{i \in A \mid a \in Act_i\}$ has all agents that can perform action $a$. The parallel composition of AMAS is given by Interleaved Interpreted Systems, which extend AMAS with propositional variables and define *global-states* and *-transitions*.

**Definition 2 (Interleaved Interpreted System [17]).** *Let $PV$ be a set of propositional variables. An* interleaved interpreted system (IIS) *is an AMAS extended with (i) a set of* global states $S \subseteq \prod_{i=1}^{n} L_i$; *(ii) an* initial state $\iota \in S$; *(iii) a* (partial) global transition function $T\colon S \times Act \to S$ *s.t.* $\forall i \in Agent(a)$, $T(s_1, a) = s_2$ *iff* $T_i(s_1^i, a) = s_2^i$, *whereas* $\forall i \in A \backslash Agent(a)$, $s_1^i = s_2^i$, *where* $s_1^i$ *is the* $i$*-th local state of* $s_1$; *and (iv) a* valuation function $V\colon S \to 2^{PV}$.

## 4 Extending the AMAS model

As defined in [17], AMAS do not include attributes. Therefore, to model ADTrees we now define *Extended AMAS*, associating attributes with local transitions.

**Definition 3 (Extended Asynchronous Multi-Agent Systems).** *An* Extended Asynchronous Multi-Agent System (EAMAS) *is an AMAS where each local transition function* $t \in LT = \bigcup_{i \in A} T_i$ *has a finite set of variables* $AT_t = \{v_t^1, \dots, v_t^k\}$ *(*attributes*) over a domain* $D_t = d_t^1 \times \cdots \times d_t^k$.

*Let* $AT = \bigcup_{t \in T} AT_t$ *and* $D = \prod_{t \in T} D_t$. *Let Guards be the set of formulæ of the form* $\beta \sim 0$, *where* $\beta$ *is a linear expression over attributes of* $AT$ *and* $\sim \in \{<, \leqslant, =, \geqslant, >\}$. *Let* $M$ *be the set of all messages,* $FUN$ *be all functions taking arguments in* $AT$*, and* $EXP(AT, FUN)$ *be linear expressions over* $AT$ *and* $FUN$. *Each transition* $t \in LT$ *has associated: (i) a* message $f_M(t) \in (\{!, ?\} \times M) \cup \{\bot\}$; *(ii) a* guard $f_G(t) \in Guards$; *(iii) an* update function $f_t\colon AT_t \to EXP(AT, FUN)$.

Item (i) indicates whether transition $t$ does not synchronise ($\bot$), or sends (marked with !) or receives (?) a message $m$. For ADTrees, $m \in M = \{ok, nok\}$. Guards in item (ii) constrain transitions. Item (iii) states how taking a transition modifies the associated attributes. To model ADTrees we further extend IIS.

**Definition 4 (Extended Interleaved Interpreted System).** *Let* $PV$ *be a set of propositional variables,* $\mathbf{v}\colon AT \to D$ *a valuation of the attributes, and* $\mathbf{v}_0$ *an initial valuation. An* extended interleaved interpreted system (EIIS)*, or a* model*, is an EAMAS extended with (i) a set of* global states $S \subseteq L_1 \times \cdots \times L_n \times D$; *(ii) an* initial state $s_0 = \langle(\iota_1, \dots, \iota_n), \mathbf{v}_0\rangle \in S$; *(iii) a* global transition relation $T \subseteq S \times Act \times S$ *s.t.* $\langle(l_1, \dots, l_n, \mathbf{v}), a, (l_1', \dots, l_n', \mathbf{v}')\rangle \in T$ *iff: either A):* $|Agent(a)| = 1 \wedge \exists t_i = (l_i, a, l_i') \in T_i$ *for* $i \in Agent(a) \wedge \forall k \in A \backslash \{i\}\ l_k = l_k' \wedge \mathbf{v} \models f_G(t_i) \wedge \mathbf{v}' = \mathbf{v}[AT_{t_i}]$; *or B):* **(a)** $\exists i, j \in Agent(a) \wedge \exists t_i = (l_i, a, l_i') \in T_i \wedge \exists t_j = (l_j, a, l_j') \in T_j$ *s.t.* $f_M(t_i) = (!, m) \wedge f_M(t_j) = (?, m)$; **(b)** $\forall k \in A \backslash \{i, j\}\ l_k = l_k'$; **(c)** $\mathbf{v} \models f_G(t_i) \wedge f_G(t_j)$; **(d)** $\mathbf{v}' = \mathbf{v}[AT_{t_i}][AT_{t_j}]$, *where* $AT_{t_i}$ *and* $AT_{t_j}$ *are disjoint; and (iv) a* valuation function $V : S \to 2^{PV}$. *In item* **(d)**, $\mathbf{v}[AT_{t_i}][AT_{t_j}]$ *indicates the substitution of attributes in the valuation* $\mathbf{v}$ *according to transitions* $t_i$ *and* $t_j$, *that is* $\mathbf{v}' = \mathbf{v}\Big[\bigwedge_{v_{t_i} \in AT_{t_i}} v_{t_i} := f_{t_i}(v_{t_i})\Big]\Big[\bigwedge_{v_{t_j} \in AT_{t_j}} v_{t_j} := f_{t_j}(v_{t_j})\Big]$.

In the definition of the global transition relation $T$, item **(a)** specifies the synchronisation of transition $t_i$ (with a sending action) and $t_j$ (with a receiving action) that share the message $m$. Item **(b)** ensures that agents other than $i$ and $j$ do not change their states in such a synchronisation. Item **(c)** guarantees that the guards of $t_i$ and $t_j$ hold for the valuation $\mathbf{v}$. Finally, item **(d)** indicates how $\mathbf{v}'$ is obtained by updating $\mathbf{v}$ with the attributes values modified by $t_i$ and $t_j$.

## 5 EAMAS transformation of ADTrees

We give formal semantics to ADTrees as EIIS. For that, we model ADTree nodes as EAMAS via *transformation patterns*. The resulting EAMAS synchronise with each other via shared actions. Note that unlike formalisms such as Timed Automata where clocks let time elapse, time in EAMAS is an attribute.

We show that this compositional approach is correct, i.e. *complete*—all relevant ADTree paths are captured by the model—and *sound*—no new paths are introduced leading to a node's goal. Moreover, these semantics are amenable to state-space reductions [21], and naturally support shared subtrees in the ADTree, all of which favours its use in practical scenarios.

### 5.1 Transformation patterns

Table 2 shows each ADTree construct transformed into one agent (sub-) model. In this compositional modelling approach, agents communicate via the blue transitions. Transformations are symmetrical for attack and defence nodes: Table 2 shows attack patterns. A leaf signals action $a$ iff it succeeds. Self-loops in states $l_A, l_N$ synchronise with all nodes that depend on this node (leaf or gate). Thus our semantics can model ADTrees where gates share children. An `AND` gate succeeds when all actions occur, in any order. Then attack $A$ occurs, followed by a broadcast of signal $!A\_ok$ by the `AND` gate. The `AND` fails if any action of a child fails. `OR`, `CAND`, and `COR` operate in the expected analogous way. Models for `SAND` and `SCAND` enforce the absence of parallelism, as per their semantics in Sec. 2.1.

To go from the initial to a final state, EAMAS patterns in Table 2 use one order of actions. We now show that this order represents all orderings of actions that make the attack/defence succeed. That is, our semantics is complete, in that it captures all paths of successful attacks/defences—see also [6].

**Theorem 1 (Completeness).** *Let $a_1, \ldots, a_n$ be the children of the ADTree gate $A$ with EAMAS models $M_{a_1}, \ldots, M_{a_n}, M_A$ resp. Let $A$ succeed when $a_{i_1} \cdots a_{i_m}$ finalise (succeeding or failing) in that order. If the EAMAS models $M_{a_{i_j}}$ finalise in the same order, then $M_A$ transits from its initial state $l_0$ to its final state $l_A$.*

*Proof.* First note that if node $x$ finalises, its EAMAS model will send *either* $!x\_ok$ or $!x\_nok$. Moreover, due to the self-loops in the end states, this happens infinitely often. Thus, if nodes $a_{i_1} a_{i_2} \cdots a_{i_m}$ finalise, actions $!a_{i_1}\_ok$, $!a_{i_2}\_ok, \ldots,$ $!a_{i_m}\_ok$ (or the corresponding $!a_*\_nok$) will be signaled infinitely often. By hypothesis, gate $A$ succeeds when $a_{i_1} \cdots a_{i_m}$ finalise in that order. All patterns in Table 2 have (at least) one sequence of actions $?a_{j_1}\_ok \cdots ?a_{j_k}\_ok$ (or $?a_*\_nok$)

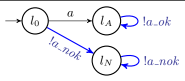
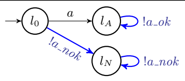
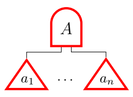
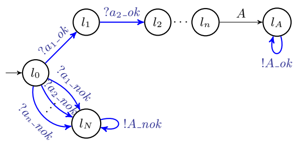
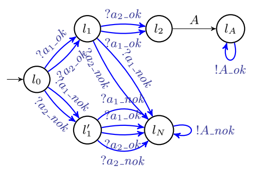
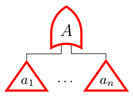
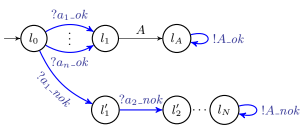
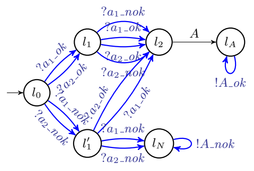
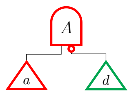
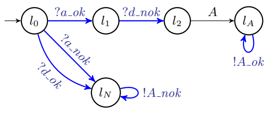
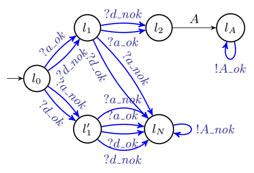
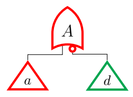
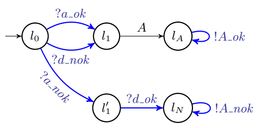
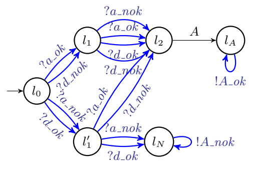
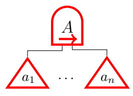
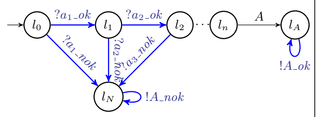
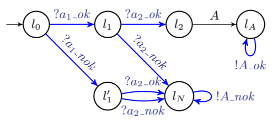
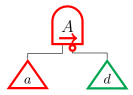
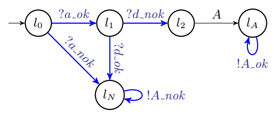
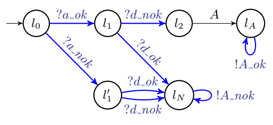
| **ADTree construct** | **Reduced Model (EAMAS)** | **Full Model (2 children only)** |
|---|---|---|
| Leaf node | | |
| |  |  |
| Conjunction/disjunction nodes | | |
|  |  |  |
|  |  |  |
| Countering nodes | | |
|  |  |  |
|  |  |  |
| Sequential nodes | | |
|  |  |  |
|  |  |  |

Table 2: ADTree nodes and corresponding agent models

that take it from $l_0$ to $l_A$. By the first argument, all actions in the sequence of $M_A$ are signaled infinitely often. $M_A$ will then transit from $l_0$ to $l_A$.

This covers *expected* actions that a parent must receive from its children to achieve its goal. For *unexpected* sequences of signals, a parent may not react to all information from its children—e.g. a `CAND` gate that after $?a\_ok$ receives (unexpectedly) $?d\_ok$. In such scenarios the model cannot reach its final state, entering a deadlock. This means that the model of $A$ cannot signal its $!A\_ok$ action. Notice that this is exactly what should happen, because such unexpected sequences actually inhibit the goal of node $A$. To formally complete this argument, we now prove that the transformations of Table 2 are sound. That is, that all paths (of actions) that make the model of a node $A$ signal $!A\_ok$, correspond to an ordered sequence of finalising children of $A$ that make it reach its goal.

**Theorem 2 (Soundness).** *Let $a_1, \ldots, a_n$ be the children of the ADTree gate $A$ with EAMAS models $M_{a_1}, \ldots, M_{a_n}, M_A$ respectively. Let the sequence of actions $?a_{i_1}\_s_{i_1}\, ?a_{i_2}\_s_{i_2} \cdots ?a_{i_m}\_s_{i_m}$ take $M_A$ from its initial state $l_0$ to its final state $l_A$, where $s_j \in \{ok, nok\}$. Then the corresponding ordered success or failure of the children $a_{i_1}, \ldots, a_{i_m}$ make the ADTree gate $A$ succeed.*

*Proof.* First, observe that the reduced models in Table 2 are subsets of the *full models*—which consider all possible interleavings of synchronisation messages from child nodes. Thus, any path $\pi$ in a reduced model also appears in the corresponding full model. Moreover, inspecting Tables 1 and 2 shows that, in the full model $\overline{M}_A$ of gate $A$, a path of actions $?a_i\_s_i$ (from children $a_i$ of $A$) that transit from $l_0$ to $l_A$, encodes the ordered finalisation of these children that make the ADTree $A$ succeed. Our hypothesis is the existence of a path $\pi$ of actions in (the reduced model) $M_A$, that take this EAMAS from $l_0$ to $l_A$. By the first argument, $\pi$ is also a path in (the full model) $\overline{M}_A$. By the second argument, $\pi$ encodes the ordered finalisation of children $c$ of $A$ that make this gate succeed.

### 5.2 Adding node attributes

Transformation patterns in Table 2 concern only an ADTree *structure*. To take the value of *attributes* into account, a child must transmit these to all parents.

**Attributes and computation functions.** Attributes attached to a node are given by the intrinsic value and computation function of the node—see Sec. 2.2 and the EAMAS update function in item (iii) of Def. 3. For example, the cost of an `AND` gate is typically its own cost plus those of all children. Attributes values are computed only after all the preconditions of the node are satisfied. `OR` gates involve a choice among children: here, computation happens upon receiving the message from one child. Conditions associated with ADTree countering nodes (e.g. `TS` in Fig. 2) are added as guards to the corresponding action transition.

**Distribution of agents over nodes.** Computation functions are a versatile solution to analyse agents coalitions. For instance, the attack time of an `AND` gate can depend on the number of agents (within the attack party) that cooperate to execute its children. This can be different for the attack *cost* of the same `AND` gate, and for the attack time of a `SAND` gate. We illustrate this in the following.

*Example 1.* In our running example—Fig. 2—the computation function for the attack time of the gate $\mathtt{ST} = \mathtt{AND}(\mathtt{b}, \mathtt{f})$ can be generically expressed with binary functions $f$ and $g$ as $f\big(\mathit{init\_time}(\mathtt{ST})\,,\, g(\mathit{init\_time}(\mathtt{b}), \mathit{init\_time}(\mathtt{f}))\big)$, where:

- $f = +$, since `ST` depends logically on the completion of both children nodes;

- $g$ depends on the agent coalition: if we use a single thief for `b` and `f` then $g = +$, if we use two thieves then $g = \max$.

This duality of $g$ is precisely the kind of analyses that our approach enables. ♦

In the general case, the children of a binary gate will be subtrees `L` and `R` rather than leaves, which allows for more complex computations of potential parallelism between the agents in `L` and `R`. For example, to model a worst-case scenario in an `AND` gate with a set of agents $A$, let $A_{\mathtt{L}} \subseteq A$ (resp. $A_{\mathtt{R}} \subseteq A$) be all agents from subtree `L` (resp. `R`). Then let the attack time of the `AND` be either:

- *the sum* of the times of `L` and `R` if $A_{\mathtt{L}} \cap A_{\mathtt{R}} \neq \varnothing$, because then some agent is used in both children and thus full parallelism cannot be ensured;
- *the maximum* between these times otherwise, since `L` and `R` are independent.

Notice that these considerations also cover cases where gates share children, e.g. $A_{\mathtt{L}} \cap A_{\mathtt{R}} \neq \varnothing$ above. The main advantage of computations functions—and our semantics in general—w.r.t. other approaches in the literature is that agents coalitions that are internal to an attack or defence party can be modelled in any desired way. In [6] and https://up13.fr/?VvxUgNCK we give more examples.

## 6 Experiments

Two state-of-the-art verification tools are used to exercise our approach and provide an empirical demonstration of its capabilities:

– Uppaal [11] is a model-checker for real-time systems. Automata templates are declared in its GUI and given semantics as Priced Time Automata. A full model is an automata network built from the instantiation of the templates, which can be queried using PCTL-like formulæ.

– IMITATOR [4] is a model-checker for Parametric Timed Automata [3], that implements full-synchronisation semantics on the shared transition labels of the automata in the model. Moreover, IMITATOR can synthesise parameter constraints e.g. to find the values of attributes so that an attack is feasible.

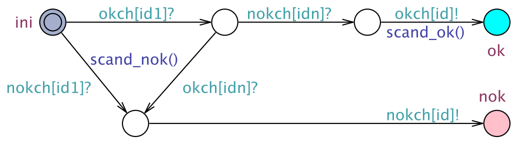


Fig. 3: Uppaal template for `SCAND` pattern

The GUI of Uppaal permits straightforward implementations of our transformation patterns: compare the pattern for the `SCAND` gate (last row in Table 2) with its corresponding Uppaal template (Fig. 3). Furthermore, the instantiation of templates to define the full model ("system declarations") makes it easy to describe the ADTree structure, as well as the agents distribution via an array. Our Uppaal models are available as XML files in https://up13.fr/?VvxUgNCK.

In turn, the open source tool IMITATOR [2] can use symbolic computations to find out the values of attributes that permit or prevent an attack. So for instance, instead of stating that the police takes 10 min to arrive, our running example could have the time $t_p$ set as a parameter variable. Checking when the attack fails results in IMITATOR outputting a set of constraints, e.g. $\{t_p \leqslant 5\}$.

We implemented the tool `adt2amas` [1] to execute our transformation process automatically. The tool receives as input the ADTree model and recursively applies the transformation rules. Then, it compiles the generated EAMAS model into the format of the tool IMITATOR. Our tool also generates a PDF file with the EAMAS model—see e.g. Fig. 5 in [6] and also https://up13.fr/?VvxUgNCK.

## 6.1 Case studies

The two above mentioned tools were used to analyse 3 literature case studies, detailed in https://up13.fr/?VvxUgNCK and [6]. These case studies were chosen so that their ADTree structures are easy to present and grasp by the readers. Notice that our approach can scale to much larger state spaces as shown in [21].

**Forestalling a software release** is based on a real-world attack to the intellectual property of a company from a competitor that wants to be "the first to market" [10]. We follow [24] where software extraction and deployment by the competitor must occur before the lawful company deploys its own product.

**Compromise IoT device** describes an attack to an Internet-of-Things device via wireless or wired LAN. The attacker accesses the LAN, acquires credentials, and then exploits software vulnerabilities to run a malicious script. Our ADTree adds defence nodes on top of the attack trees used in [25].

**Obtain admin privileges** models a hacker trying to escalate privileges in a UNIX system, via physical access to an already logged-in CLI or remote access (attacking SysAdmin). This well known case study [29, 19, 23, 24] has a branching structure: all gates but one are `OR` in the original tree of [29]. We enrich this with the `SAND` gates of [24], and further add reactive defences.

## 6.2 Experimentation setup

We computed the cost and time of attacks of four ADTrees: one corresponding to our running example, plus one for each case study. Each tree was implemented: 1) in IMITATOR using the patterns of Table 2 (we call this "EAMAS"); 2) in UPPAAL using the same patterns (also "EAMAS"); and 3) in UPPAAL using the original templates of [24, 25] that employ clocks and time constraints, extended to fit Sec. 6.1 ("ORIGIN"). Such triple implementation pursues two goals:

(a) *verify correctness*, checking that the results of reachability ("can the attack succeed?") and quantitative queries ("what is the time of the fastest possible attack?", "what is the lowest cost incurred?") coincide between the ORIGIN and EAMAS implementations, regardless of the tool used;

(b) *demonstrate our contributions*: studying the impact of agent coalitions on quantitative metrics such as minimal attack time/cost; and synthesising the parameter valuations rendering an attack feasible.

## 6.3 Verifying correctness

To achieve goal (a) we queried the min/max time and cost of attacks for each ADTree: Table 3 summarises our results. For all ADTrees, all queries on the EAMAS implementations in IMITATOR and in UPPAAL yielded the same values, thus the joint name "EAMAS." In the running example, adding the constraint on

|  |  | Time of an attack |  |  |  | Cost of an attack |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  |  | min |  | max |  | min |  | max |  |
| Number of attack agents: |  | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| treasure hunters | ORIGIN | – | 125 min | – | 132 min | – | € 690 | – | € 1190 |
|  | EAMAS | 185 min | 125 min | 185 min | 125 min | € 1100 | € 1190 | € 1100 | € 1190 |
| forestall | ORIGIN | – | 43 days | – | 55 days | – | € 4k | – | € 10.5k |
|  | EAMAS | 43 days | 43 days | 55 days | 55 days | € 4k | € 4k | € 7.5k | € 7.5k |
| iot-dev | ORIGIN | – | 694 min | – | 695 min | – | € 270 | – | € 380 |
|  | EAMAS | 784 min | 694 min | 1114 min | 694 min | € 270 | € 270 | € 320 | € 320 |
| gain-admin | ORIGIN | – | 2942 min | – | 23070 min | – | € 100 | – | € 15820 |
|  | EAMAS | 2942 min | 2942 min | 23070 min | 23070 min | € 100 | € 100 | € 6000 | € 6000 |

Table 3: Quantitative results for ADTrees implementations: ORIGIN vs. EAMAS

the police (associated with TS) would impact the structure of the ORIGIN model too much, and thus this constraint was only implemented in the EAMAS versions.

Six values in Table 3 (underlined) differ between the EAMAS and ORIGIN implementations of an ADTree. For max cost this is because the ORIGIN models consider all possible actions for maximisation, unnecessary for e.g. OR gates. An EAMAS model by default considers a single attack in such cases, but it can mimic ORIGIN if one forces the occurrence of all attacks (even for OR gates).

The correct max time of iot-dev with 2 agents is the one for EAMAS: 694 min, via attacks CPN, GVC, esv, and rms. We suspect that the ORIGIN UPPAAL implementation yields an imprecise result due to the use of non-discrete clocks and our remark *i)* on UPPAAL's time abstractions when clock variables are used.

In treasure hunters, the time condition for TS (see Fig. 2) is much more difficult to model in UPPAAL with clocks than in the EAMAS model where time is an attribute. The value €690 in the ORIGIN model is incorrect because it uses e (emergency gate), which would be too slow to get away from an alerted police.

Table 3 also shows initial experiments with agents: we used one or two attackers, in the latter case setting the time-optimal agent distribution. This distribution was chosen according to the structure of the tree; so for instance when two attackers are used in the iot-dev case study, we set different agents for the leaves of gate APN, allowing attacks CPN and GVC to run in parallel.

Such agents coalitions were easily encoded (as arrays) in the EAMAS models. In contrast, the ORIGIN ADTree implementations use clock variables and constraints to encode the duration of attacks/defences. This approach—standard in verification of real time systems—has two drawbacks when analysing ADTrees:

*i)* UPPAAL uses abstractions and approximations for time zones that rule out decidability in the general case. Thus, queries (e.g. "is an attack feasible?") result in "may be true/false." In contrast, EAMAS transformations are untimed and verification is exact. The price to pay is a larger state space (which we did not reduce but see [21]) and approx. thrice-longer computation times.

*ii)* Unless explicitly encoded in each gate, time elapses simultaneously for all clocks. This is equivalent to having an agent for each tree node. Thus, modelling dependent node executions requires e.g. using a SAND gate rather than an AND gate. This contaminates the structure of the ADTree with the distribution of the agents that perform the actions. In contrast, EAMAS can keep the ADTree structure unmodified while studying agents coalitions.

Remark *ii)* makes it impossible to analyse one-agent coalitions in the ORIGIN implementations. Therefore, for each ADTree of Table 3, ORIGIN entries only have results for the maximum number of time-parallelisable agents in that tree.

## 6.4 Playing with agents coalitions

In the running example, the min/max times differ for one and two agents, and an extra cost (€ 90) is incurred when the second agent is used. In contrast, the forestall and gain-admin case studies are intrinsically sequential, hence attack times (and cost) are unaffected by the number of agents.

The iot-dev case study behaves similar to the running example when adding an extra agent. However, the minimal time decreases when an agent handles the `CPN` subtree while the other one is assigned to `gc`. Since `gc` has a longer duration than any option in the `CPN` subtree, the choice it makes does not change the operation time. With one agent, both `gc` and an option of `CPN` are achieved by the same agent, leading to different min and max times, depending on the choice.

Fig. 4 shows attack times for a different ADTree (that can be found in https://up13.fr/?VvxUgNCK and also in [6]) where changing the agents coalition has a larger impact in attack metrics. The chart shows the fastest and slowest attack times achieved with different assignments of agents to nodes, where all nodes take 1 time unit to complete.

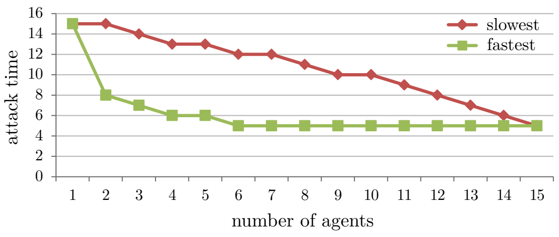


Fig. 4: Scaling w.r.t. the assignment of agents

These times coincide when there is a single agent, or one agent per node, since then there is only one way to assign agents to nodes. Instead, in the middle cases, the difference between fastest and slowest attack varies substantially for different agents coalitions. Such difference would be exacerbated by more heterogeneous time attributes in the nodes. The analyses enabled by our approach show that the fastest attack can be achieved using only 6 agents.

## 6.5 Parameter synthesis

We also experimented with the parametric capabilities offered by IMITATOR:

**Treasure hunters**: "To catch the thieves, what is the maximum time the police can take to arrive?" Answering this question requires synthesising a value for the time attribute of the `p` node, which becomes a model *parameter*. IMITATOR computed that the police can take at most 5 minutes to prevent the burglary.

**Case studies**: an attack is successful if its associated defence was slower. (*i*) for forestall, `id` should take at most 1 day to block `NA`—since `NAS` is a *failed reactive defence*, `id` is triggered as soon as `heb` succeeds, and must finish faster than the intrinsic time of `NA`; (*ii*) for iot-dev, `inc` is effective iff it takes at most 3 min; (*iii*) for gain-admin we proved that whatever the time for `tla`, an attack is feasible (as `GSAP` is a disjunction), hence the other defences are required.

## 7 Conclusion and future work

We revisited Attack-Defence Trees under a unified syntax, extending the usual constructs with a new sequential counter-operator (`SCAND`). More importantly we introduced EAMAS, an agent-aware formalism to model ADTrees, and transformation patters from the latter to the former that are sound, complete, and preserve the compositionality of ADTrees, naturally covering cases with shared subtrees. The impact of different agent coalitions on attack time and cost was evaluated using UPPAAL and IMITATOR. Finally, the feasibility of an attack was evaluated through parameter synthesis with IMITATOR, to obtain the attribute values of ADTree nodes that make an attack succeed. Our experiments show that (and how) different agent distributions affect the time of attacks/defence strategies, possibly rendering some infeasible. We expect this will open the gate to richer studies of security scenarios, with multiple agents that can collaborate.

Our next goals include logics to express properties in EAMAS, and adapting the *partial order reduction* from [17] as well as the *state space reduction for tree topologies* of [21] to agent strategies in EAMAS, including extensions to parametric timing information. This will allow for studying the strategic abilities of agents, ultimately in a parametric setting. Finally, we will add support for agents assignment to our tool `adt2amas` that transforms ADTrees into EAMAS.

## References

1. ADT2AMAS. https://depot.lipn.univ-paris13.fr/parties/tools/adt2amas
2. IMITATOR. https://www.imitator.fr
3. Alur, R., Henzinger, T., Vardi, M.: Parametric real-time reasoning. In: ACM Symposium on Theory of Computing, 1993. pp. 592–601. ACM (1993). https://doi.org/10.1145/167088.167242
4. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In: FM 2012. LNCS, vol. 7436, pp. 33–36. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_6
5. André, É., Lime, D., Ramparison, M., Stoelinga, M.: Parametric analyses of attack-fault trees. In: ACSD 2019. pp. 33–42. IEEE (2019). https://doi.org/10.1109/ACSD.2019.00008
6. Arias, J., Budde, C.E., Penczek, W., Petrucci, L., Sidoruk, T., Stoelinga, M.: Hackers vs. security: Attack-Defence Trees as Asynchronous Multi-Agent Systems. HAL (2020), https://hal.archives-ouvertes.fr/hal-02902348
7. Arnold, F., Guck, D., Kumar, R., Stoelinga, M.: Sequential and parallel attack tree modelling. In: SAFECOMP. pp. 291–299. Springer (2015). https://doi.org/10.1007/978-3-319-24249-1_25
8. Aslanyan, Z., Nielson, F.: Pareto Efficient Solutions of Attack-Defence Trees. In: Principles of Security and Trust, vol. 9036, pp. 95–114. Springer (2015). https://doi.org/10.1007/978-3-662-46666-7_6
9. Aslanyan, Z., Nielson, F., Parker, D.: Quantitative Verification and Synthesis of Attack-Defence Scenarios. In: CSF 2016. pp. 105–119. IEEE (2016). https://doi.org/10.1109/CSF.2016.15
10. Buldas, A., Laud, P., Priisalu, J., Saarepera, M., Willemson, J.: Rational Choice of Security Measures Via Multi-parameter Attack Trees. In: Critical Information Infrastructures Security. pp. 235–248. Springer (2006)

11. David, A., Larsen, K., Legay, A., Mikučionis, M., Poulsen, D.: Uppaal SMC tutorial. STTT **17**(4), 397–415 (2015). https://doi.org/10.1007/s10009-014-0361-y
12. Fagin, R., Halpern, J., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT Press (1995)
13. Fila, B., Widel, W.: Efficient attack-defense tree analysis using Pareto attribute domains. In: CSF 2019. pp. 200–215. IEEE (2019). https://doi.org/10.1109/CSF.2019.00021
14. Gadyatskaya, O., Hansen, R., Larsen, K., Legay, A., Olesen, M., Poulsen, D.: Modelling Attack-defense Trees Using Timed Automata. In: FORMATS 2016, LNCS, vol. 9884, pp. 35–50. Springer (2016). https://doi.org/10.1007/978-3-319-44878-7_3
15. Gribaudo, M., Iacono, M., Marrone, S.: Exploiting Bayesian networks for the analysis of combined attack trees. ENTCS **310**, 91–111 (2015). https://doi.org/10.1016/j.entcs.2014.12.014
16. Hermanns, H., Krämer, J., Krčál, J., Stoelinga, M.: The Value of Attack-Defence Diagrams. In: POST 2016. LNCS, vol. 9635, pp. 163–185. Springer (2016)
17. Jamroga, W., Penczek, W., Dembinski, P., Mazurkiewicz, A.: Towards partial order reductions for strategic ability. In: AAMAS 2018. pp. 156–165. ACM (2018)
18. Jhawar, R., Kordy, B., Mauw, S., Radomirović, S., Trujillo-Rasua, R.: Attack trees with sequential conjunction. In: ICT Systems Security and Privacy Protection. pp. 339–353. Springer (2015). https://doi.org/10.1007/978-3-319-18467-8_23
19. Jürgenson, A., Willemson, J.: Computing Exact Outcomes of Multi-parameter Attack Trees. In: OTM 2008. pp. 1036–1051. Springer (2008)
20. Khand, P.: System level security modeling using attack trees. In: 2nd International Conference on Computer, Control and Communication. pp. 1–6 (2009). https://doi.org/10.1109/IC4.2009.4909245
21. Knapik, M., Penczek, W., Petrucci, L., Sidoruk, T.: Squeezing state spaces of (Attack-Defence) trees. In: ICECCS 2019. IEEE Computer Society (2019)
22. Kordy, B., Mauw, S., Radomirović, S., Schweitzer, P.: Attack–defense trees. Journal of Logic and Computation **24**(1), 55–87 (2014). https://doi.org/10.1093/logcom/exs029
23. Kordy, B., Piètre-Cambacédès, L., Schweitzer, P.: DAG-based attack and defense modeling: Don't miss the forest for the attack trees. Computer Science Review **13-14**, 1–38 (2014). https://doi.org/10.1016/j.cosrev.2014.07.001
24. Kumar, R., Ruijters, E., Stoelinga, M.: Quantitative attack tree analysis via priced timed automata. In: FORMATS. LNCS, vol. 9268, pp. 156–171. Springer (2015)
25. Kumar, R., Schivo, S., Ruijters, E., Yildiz, B., Huistra, D., Brandt, J., Rensink, A., Stoelinga, M.: Effective analysis of attack trees: A model-driven approach. In: Fundamental Approaches to Software Engineering. pp. 56–73. Springer (2018)
26. Kumar, R., Stoelinga, M.: Quantitative security and safety analysis with attack-fault trees. In: HASE 2017. pp. 25–32. IEEE (2017)
27. Lomuscio, A., Penczek, W., Qu, H.: Partial order reductions for model checking temporal epistemic logics over interleaved multi-agent systems. In: AAMAS 2010. vol. 1-3, pp. 659–666. IFAAMAS (2010)
28. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.: Automated generation and analysis of attack graphs. In: Proceedings 2002 IEEE Symposium on Security and Privacy. pp. 273–284 (2002). https://doi.org/10.1109/SECPRI.2002.1004377
29. Weiss, J.: A system security engineering process. In: Proceedings of the 14th National Computer Security Conference. pp. 572–581 (1991)
30. Widel, W., Audinot, M., Fila, B., Pinchinat, S.: Beyond 2014: Formal methods for attack tree-based security modeling. ACM Comp. Surv. **52**(4), 75:1–75:36 (2019). https://doi.org/10.1145/3331524