A Template-Based Method for the Generation of Attack Trees

Jeremy Bryans[1], Lin Shen Liew[2], Hoang Nga Nguyen[1(✉)], Giedre Sabaliauskaite[2], Siraj Shaikh[1], and Fengjun Zhou[2]

[1] Coventry University, Coventry, UK
{ac1126,ac1222,aa8135}@coventry.ac.uk
[2] Singapore University of Technology and Design, Singapore, Singapore
{linshen_liew,giedre,fengjun_zhou}@sutd.edu.sg

Abstract. Attack trees are used in cybersecurity analysis to give an analyst a view of all the ways in which an attack can be carried out. Attack trees can become large, and developing them by hand can be tedious and error-prone. In this paper the automated generation of attack trees is considered. The method proposed is based on a library of attack templates – parameterisable patterns of attacks such as denial of service or eavesdropping – and that also uses an abstract model of the network architecture under attack. A pseudocode implementation of the method is also presented. The example application given is from the automotive domain and using an architecture consisting of linked CAN networks – a network configuration found in virtually every current vehicle.

Keywords: Attack trees · Generation · Automotive · Cybersecurity

1 Introduction

Attack trees are a well-known graphical model for capturing and analysing attacks on a system [12]. Their intuitive simplicity and ability to succinctly capture all attacks on a system have made them popular in many domains, including SCADA systems [2], ATM security [4], the analysis of insider attacks [11] and the automotive domain [1]. They give an analyst an overview of all the known ways in which an attack can be carried out, and show how single attack steps combine and build into complex attacks.

Attack trees are directed acyclic graphs with a single end node, which is the goal of the attack. To construct an attack an analyst considers all the steps which would immediately lead to the goal of the attack being realised. These become the subgoals, or intermediate leaves of the tree. Each of these leaves is now considered as a (sub)goal, and the steps that would lead to it's realisation are identified. The process is recursively repeated until the branches of the tree cannot be further expanded. This process can be time-consuming, especially for large attack trees [4] and several researchers have therefore investigated the automatic generation of the trees [5,6,10,13].

Published by Springer Nature Switzerland AG 2020
M. Laurent and T. Giannetsos (Eds.): WISTP 2019, LNCS 12024, pp. 155–165, 2020.
https://doi.org/10.1007/978-3-030-41702-4_10

Within the literature, two main approaches to automating the generation or synthesis of attack trees have developed: (i) model transformation and (ii) semantic-based construction. In the model transformation approach, a target system and attackers are modelled using either graphical [6] or formal [13] presentations as input. The desired target of the attackers is identified, and from this the tree root from which the tree construction starts is established. In [6], system models contain actors, processes, items and locations, and connections between these elements to the desired target are utilised to develop the attack tree. Similarly, systems and attackers in [13] are modelled in a process calculus as input. They are first transformed into propositional formulae. Given a target location, these formulae are utilised to construct attack trees by means of backwards-chaining search. While techniques in the model transformation approach are automated, they suffer from lacking a basis for correctness. There is no rigorous relation between generated attack trees and the attacks implicitly implied from input models. In order to fill this gap, [10] proposed ATSyRA, an interactive tool for synthesising attack trees from attack graphs. First, ATSyRA generates all attack paths from the input graphs by model checking. Then, users are required to specify a refinement relation between a set of actions to recursively refine attack paths to eventually construct an attack tree. While ATSyRA establishes the semantic connection between the constructed tree and the input model via attack paths, it is not fully automated. To overcome this shortcoming, [8] introduced an approach to extending an existing attack tree by means of a library of attack trees. The extension is enabled by adding logical preconditions and assertions to tree nodes. Then an attack tree from the library can be attached to a node of the attack tree to be extended if certain relations between the preconditions and assertions are satisfied. To this end, logical reasoning must be employed. Similarly, [5] has proposed a different approach which is based on the formal semantics of attack trees [7]. To this end, the synthesis problem becomes that of generating attack trees from a given semantics, i.e., a set of attack traces. It is reduced to a biclique problem, which is known to be NP-complete, and a heuristic algorithm is suggested for the construction.

In this paper we propose a method for the generation of attack trees based on *templates*: abstracted and parameterizable known patterns of attack, and represent steps such as spoofing of one node by another, or eavesdropping on traffic between two nodes, which together can be built up into an attack. The method takes as input a description of the architecture of the network that is being attacked and the set of templates.

These networks are modelled by graphs consisting of nodes and connectivity information. Each node represents a component of the network. The network information required includes the *access points*. These are the nodes within the network that are exposed to attackers outside the network. We present a method that applies each element from the library of attack patterns to the graphical network model in order to form attack trees. We give as well an algorithm for our method. Given a network and a set of templates, the algorithm can generate all possible attacks conforming to the template library.

The contributions in this paper are the template-based method for the generation of attack trees and it's algorithm, and the automotive example demonstrating the method. The paper proceeds as follows: Sect. 2 begins with an introduction to automotive communication networks and attack trees. In Sect. 3 we give the description of the template-based methodology for generating attack trees, and in Sect. 4 we give the pseudo-code description of the generation algorithm and briefly present the results of our automotive example.

2 Background

2.1 Automotive Communication Network

An automotive communication network facilitates the communication between electronic control units (ECUs) within a vehicle. It is usually divided into subnetworks of related ECUs. Depending on the communication requirements of each subnetwork (such as bandwidth, time, etc.), different network types can be employed such as CAN, CANFD, FLEXRAY, LIN, ETHERNET, etc. These networks can be interconnected via Gateway ECUs which will coordinate the traffic between them.

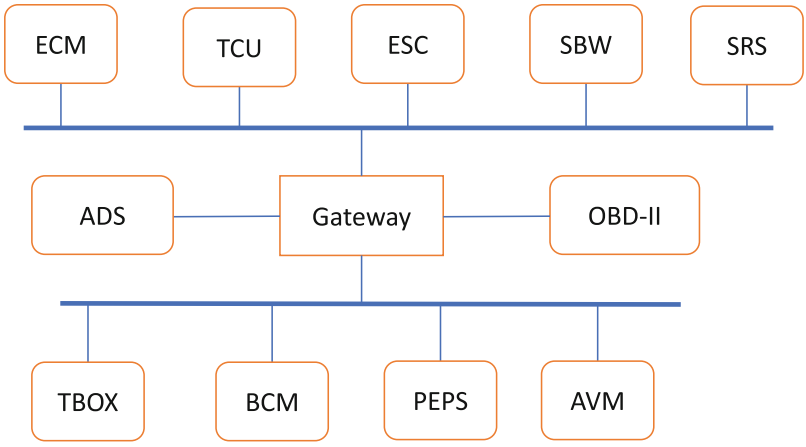

Fig. 1. An automotive internal network.

In this paper, we model an automotive communication network as a tuple $(\mathrm{NET}, \mathrm{ECU}, \mathrm{AP}, \mathrm{net})$ where NET is a finite set of subnetworks, ECU is a finite set of ECUs, $\mathrm{AP} \subseteq \mathrm{ECU}$ identifies ECUs that are accessible to attackers (such as OBD-II or TBOX), and $\mathrm{net} : \mathrm{NET} \rightarrow \wp(\mathrm{ECU})$ is a mapping to determine to which subnetwork an ECU belongs. For example, the network in Fig. 1 is modelled by $M_f = (\mathrm{NET}_f, \mathrm{ECU}_f, \mathrm{AP}_f, \mathrm{net}_f)$ where:

- $\text{NET}_f = \{\text{CAN}_1, \text{CAN}_2, \text{CAN}_3, \text{CAN}_4\}$;
- ECU_f = {ECM, TCU, ESC, SBW, SRS, ADS, Gateway, OBD-II, TBOX, BCM, PEPS, AVM};
- AP_f = { OBD-II, TBOX};
- net_f = $\{\text{CAN}_1 \mapsto \{\text{ECM, TCU, ESC, SBW, SRS, Gateway}\}, \text{CAN}_2 \mapsto \{\text{TBOX, BCM, PEPS, AVM, Gateway}\}, \text{CAN}_3 \mapsto \{\text{ADS, Gateway}\}, \text{CAN}_4 \mapsto \{\text{OBD-II, Gateway}\}\}$.

2.2 Attack Trees

Attack trees contain a goal (the root of the tree), a set of sub-goals, structured using the operators conjunction (**AND**) and disjunction (**OR**), and leaf nodes, which represent atomic attacker actions. The **AND** nodes are complete when all child nodes are carried out and the **OR** nodes are complete when at least one child node is complete.

Extensions have been proposed using **Sequential AND** (or **SAND**) [7]. We follow the formalisation of attack trees given in [7,9]. If $\mathbb{A}$ is the set of possible atomic attacker actions, the elements of the attack tree $\mathbb{T}$ are $\mathbb{A} \cup \{\mathbf{OR}, \mathbf{AND}, \mathbf{SAND}\}$, and an attack tree is generated by the following grammar, where $a \in \mathbb{A}$:

$$t ::= a \mid \mathbf{OR}(t, \ldots, t) \mid \mathbf{AND}(t, \ldots, t) \mid \mathbf{SAND}(t, \ldots, t)$$

Attack tree semantics have been defined by interpreting the attack tree as a set of series-parallel (SP) graphs [7].

3 Methodology

We develop a method to generate attack trees from a network model and a library of attack tree templates. Attack tree templates are building-blocks to assemble an attack tree. Each template from the library represents an attack step within the network which can be applied to different subnetworks and/or ECUs. The adaptability of the attack to various subnetworks and ECUs can be captured by using variables within the template. When the templates are fully instantiated with concrete values from the sets of the network model, it provides a concrete example of an attack on the network.

For example, attacks on a communication network can be categorised into two passive or active attacks; eavesdropping and traffic analysis are two examples of passive attacks, while spoofing, replay and DoS (Denial of Service) are active attacks. This is captured in Fig. 2. Variables are used in all the leaves of this template which can be replaced by concrete values. Let us consider the leaf Eavesdrop X:NET. The variables X can be replaced by any value from the component NET of the network model. If we consider the network model M_f as depicted in Fig. 1, X can be replaced by CAN_1, CAN_2, CAN_3 or CAN_4.

The connectivity between ECUs within the network will be represented in attack tree templates using lists. When instantiated, a list of ECUs corresponds

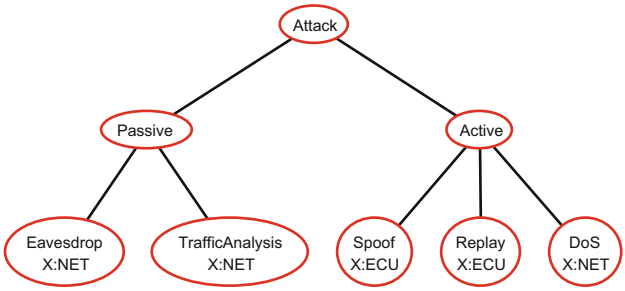

Fig. 2. An initial attack template tree.

to the ability to send data from the first ECU in the list to next one, then the next one, and so on until the data reaches the last ECU in the list. This means consecutive ECUs in the list must belong to the same subnetwork. Gateway ECUs may appear in the list to capture the connectivity between ECUs of different subnetworks. For example, if we consider the model M_f, a list of ECUs is [ECM, TCU, Gateway, TBOX] where ECM is connected to TCU and TCU to Gateway in CAN_1, and Gateway to TBOX in CAN_2.

The generation of attack trees starts with a specified template from the library. This template has no closed variables. The generation is carried out recursively. At each recursion, a leaf which may contain open variables is considered for expansion. When there are $n > 0$ assignments for the open variables, this leaf node is converted into an OR node with n children with each child corresponding to one assignment. The assignments are copies of the leaf node where the open variables are replaced by values. Each child is then replaced by a template from the library where the name of the template root matches the name of the child and the parameters of the root can be unified with the parameters of the child. The unification of the parameters will give rise to an assignment of closed variables of the template. The replacement of the child with the template will also replace all closed variables with the values from the assignment. This process is illustrated in Fig. 3. A white circle represents a node with variables while a black one states that its variables have been replaced with values by some assignment.

A special case of assignments is for unassigned lists. An assignment for an unassigned list [X .. Y] is a list of constants from NET and ECU. The start and the end of the list must satisfy any condition for X and Y. For example, consider the network M_f. An unassigned list [ECM..Y:AP] must be assigned to a list of ECUs from ECM to an ECU that is an access point, i.e, in AP_f. There are two ECUs that Y can be assigned to: OBD-II and TBOX. Then, one of the list of connected ECUs that [ECM..Y:AP] can be assigned to is [ECM, TCM, Gateway, TBOX] where they are consecutively connected and the last ECU (TBOX) is an access point. Obviously, this is not the only assignment. Two of other candidates to assign this list to are [ECM, Gateway, TBOX] and [ECM, TCM, Gateway, OBD-II].

Assigned lists [X|Y] recursively describe a list with X as the head of the list and Y as the remaining elements, i.e., the tail of the list. [] stands for an empty list. An assigned list [X|Y] normally appears at the root of some templates. When it is unified with a list of elements, X will be unified with the head and Y will be unified with the tail. For example, if the list [ECM, TCM, Gateway, TBOX] is unified with [X|Y], then X = ECM and Y = [TCM, Gateway, TBOX].

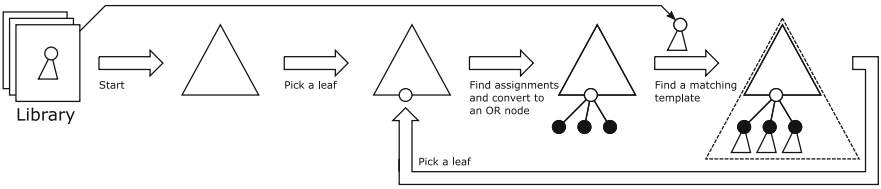

Fig. 3. Methodology.

3.1 Attack Tree Templates

More formally, nodes in an attack tree template may contain parameters which are made of variables, list terms or constants (i.e., elements of NET and ECU of a network model). Variables can be instantiated with node names. Let N be a set of names for tree nodes, V a set of variables and $C = \text{NET} \cup \text{ECU}$ a set of constants. The syntax of an attack tree template is defined below:

$$
\begin{aligned}
\mathit{tree} &::= \mathit{leaf\text{-}node} \mid \\
&\quad\ \mathit{tree\text{-}node}\mathbf{AND}(\mathit{tree}, \ldots, \mathit{tree}) \mid \\
&\quad\ \mathit{tree\text{-}node}\mathbf{SAND}(\mathit{tree}, \ldots, \mathit{tree}) \mid \\
&\quad\ \mathit{tree\text{-}node}\mathbf{OR}(\mathit{tree}, \ldots, \mathit{tree}) \\
\mathit{leaf\text{-}node} &::= n\, \mathit{parameter}^* \\
\mathit{tree\text{-}node} &::= n\, \mathit{parameter}^* \\
\mathit{parameter} &::= \mathit{variable} \mid \mathit{list} \mid c \\
\mathit{variable} &::= X[\text{“:”}\mathit{type}][/Y[\text{“:”NET}]][\#Z[\text{“:”ECU}]] \\
\mathit{type} &::= \text{NET} \mid \text{ECU} \mid \text{AP} \\
\mathit{list} &::= \mathit{unassigned\text{-}list} \mid \mathit{assigned\text{-}list} \\
\mathit{unassigned\text{-}list} &::= [\,\mathit{variable}\ \text{“..”}\ \mathit{variable}\,] \\
\mathit{assigned\text{-}list} &::= [\,\mathit{variable}\ \text{“|”}\ \mathit{variable}\,]
\end{aligned}
$$

where $X, Y \in V$, $n \in N$ and $c \in C$.

Informally, an attack tree template is an attack tree in which each node contains a name and possibly a list of parameters. A parameter can be a variable, a constant (node names) or a list of variables and constants. Variables occurring in the root node of an attack tree template are called closed variables. They may reoccur in the descendants of the root. Once root variables are instantiated, their values are propagated down to the descendant nodes correspondingly. In contrast to closed variables, variables in a template that do not appear in its root are called open.

We postulate the following conditions on the occurrence of variables on an attack tree template:

- Assigned lists can only appear at the root;
- Open variables can only appear at the leaves;
- Unassigned lists can only appear at the leaves.

The assignment of values to variables can be restricted with types, by using the condition ": type". This condition restricts a variable to be instantiated with a constant of type NET, ECU or AP. For example, consider the network in Fig. 1. Given X:NET, X can only be assigned to CAN_1, CAN_2, CAN_3 or CAN_4. Given X:ECU, X can only be assigned to ECM, TCU, Gateway, OBD-II, BCM or TBOX. AP stands for access points OBD-II and TBOX, i.e., places where attackers can have cyber access to the network. Then, X:AP says that X can only be assigned to OBD-II or TBOX. A further restriction can be introduced to the assignment by "/ Y : NET". Once Y is instantiated with a constant of type NET, "X / Y:NET" states that X can only be assigned to an ECU within the subnetwork Y. For example, " X / Y:NET" where Y is CAN_1 means that X can only be assigned to ECM, TCU, or Gateway. Finally, one can require that X is not assigned to an ECU by using the restriction $\#Z$ where Z is of type ECU. Once Z is instantiated with an ECU, X cannot be assigned to that ECU.

3.2 A Simple Example

We illustrate our method on an automotive network, depicted in Fig. 4(a). It contains two CAN buses: the powertrain, consisting of three ECUs: ECM (Engine Control Module), TCU (Transmission Control Unit) and GW (the Gateway) and the telematics bus, containing two ECUs: TBox (Telematics Box) accessible to attackers and the same GW, which connects the two buses.

This network is modelled by a tuple $(\mathrm{NET}_m, \mathrm{ECU}_m, \mathrm{AP}_m, \mathrm{net}_f)$ where:

- $\mathrm{NET}_m = \{\mathrm{CAN}_1, \mathrm{CAN}_2\}$;
- $\mathrm{ECU}_m = \{\mathrm{ECM}, \mathrm{TCU}, \mathrm{GW}, \mathrm{TBOX}\}$;
- $\mathrm{AP}_m = \{\mathrm{TBOX}\}$; and
- $\mathrm{net} = \{\mathrm{CAN}_1 \mapsto \{\mathrm{ECM}, \mathrm{TCU}, \mathrm{GW}\}, \mathrm{CAN}_2 \mapsto \{\mathrm{GW}, \mathrm{TBOX}\}\}$.

We then consider a library of attack tree templates that focus on how to compromise ECM. The library consists of two templates, depicted in Fig. 4(b) and (c). The template (b) describes a compromise attack on ECM. Essentially, this attack can be realised by starting compromising an ECU to which attackers have access to (Z:AP). Then, the compromise attack can be propagated to the next ECU connected to a compromised one until we reach ECM. This is described by the unassigned list [Z:AP .. ECM]. The template (b) is also specified as the start tree of the generation process. The template (c) describes how compromise attack can be carried out from the first ECU to the last in the list [Z|L]. Note that Z is the head of the list and L is the tail. On Fig. 4(c) the arrow between the edges leading to the two nodes indicates that both nodes must be carried

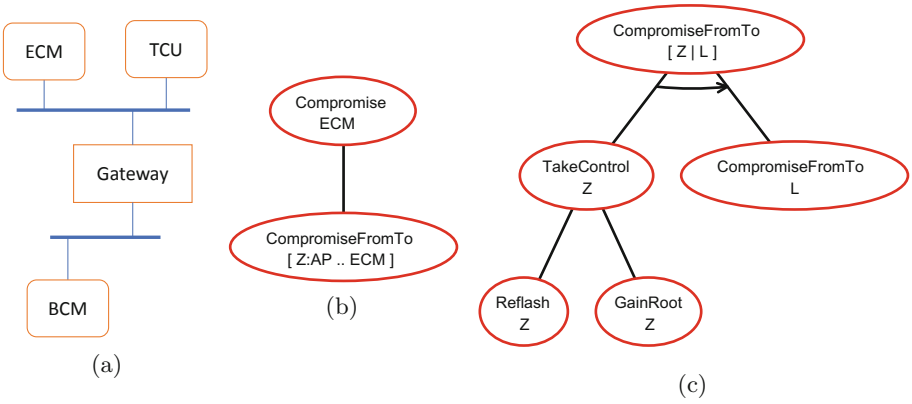

Fig. 4. The compromise attack template tree.

out in order (a **SAND** node). Not joining the edges in Fig. 4(c) signifies an **OR** node, and joining edges with a line (rather than an arrow) signifies that the root node is an **AND** node.

This is done by taking control of the ECU Z at the head of the list and then recursively taking control of the rest of the list. Taking control can be done by either re-flashing or gaining root access to Z.

Initially, the construction starts with the template (b) in Fig. 4. The leaf of this expanded tree "CompromiseFromTo [Z:AP .. ECM]" is now considered for further expansion. It has an open parameter which is an unassigned-list. There are two possible assignments for it; one is [ECM, GW, TBOX] and the other is [ECM, TCU, GW, TBOX]. However, the second list is considered redundant as ECM is directly connected and can communicate with GW without using TCU. This is derived from the nature of CAN bus communication where ECUs on the same bus are directly connected with each other. Therefore, the leaf is appended with one child corresponding to the assignment of [Z:AP .. ECM] to [ECM, GW, TBOX]. This child is then expanded by the template (c) in Fig. 4. This template is used several times depending on the length of the list. Finally, we obtain the tree[1] which has height 9 and contains 17 nodes.

4 Implementation

We now present the algorithm used to implement our generation method (Algorithm 1.) The inputs are (1) a model of the network structured as in Sect. 2 and (2) a library of attack tree templates and it produces an attack tree as the output.

The algorithm starts with the initial tree InitTree from the input library in line 2. It then loops as long as there is a leaf on the constructed tree and a

[1] The tree can be viewed at https://tinyurl.com/s55u7qh.

Algorithm 1. Generating attack trees

```
1: function BuildTree(Model, Library)
2:     tree ← InitTree ∈ Library
3:     while ∃leaf ∈ tree, subtree ∈ Library: leaf matches subtree do
4:         assignments ← getAssignments(leaf, Model)
5:         Turn leaf into an "or" nodes
6:         for each assignment of assignments do
7:             assignedLeaf ← apply(assignment, leaf)
8:             unification ← unify(subtree, assignedLeaf)
9:             add apply(unification, subtree) as a child of leaf
10:        end for
11:    end while
12:    return tree
13: end function
```

template, namely subtree, from the library that can be matched. In this loop, all assignments for the variables of the leaf are first computed in line 4. Then for each of the assignments, a unification of subtree and the application of the assignment to the leaf is calculated in line 8. Then the subtree to which the unification is applied is added as a child of the leaf in line 9. Note that the leaf is now converted into an "or" node in line 5. The loop at line 3 will continue until no more leaves and matching templates can be found.

The function apply replaces attack tree template variables with the corresponding values in the input assignment, from the root to the leaves recursively. unify in line 8 is a standard unification procedure. It tries to unify the root of subtree with the leaf to which the considered assignment is applied. It yields a unifier which can be considered as an assignment to the whole subtree.

getVarAssignments generates Cartesian product of all assignments for the variables and unassigned lists in the input leaf.

Experiment

We briefly present the experimental result of our implementation on two examples, implemented in Python[2] and carried out on a PC with a processor Intel Core i5-4590 3.3 GHz with 8GB of memory.

We first rerun the mini example described in Sect. 3.2 which confirms the output tree obtained in Sect. 3.2. Using Python "cProfile" module, the run-time of this experiment is 0.025s and uses 19739 function calls. The second experiment[4] is to generate an attack tree for the automotive network M_f as depicted in Fig. 1. It consists of 4 CAN bus networks with 12 ECUs. The template library contains 21 attack tree templates, including the initial tree as depicted in Fig. 2. In total, the run-time is 0.292s, using 574133 function calls. The generated attack tree[3] has 3756 nodes and of height 19. An attack example extracted from the tree is an eavesdropping attack carried out at a compromised TCU.

[2] The source code can be downloaded from https://tinyurl.com/uoptgfb.
[3] The tree can be viewed at https://tinyurl.com/vzscydf.

Access was gained at the TBOX, then the gateway was compromised followed by the TCU:
GainRoot(TBOX) → Reflash(GW) → Reflash(TCU) → CollectDataFrom(TCU).

5 Conclusion

In this paper, we have proposed a practical method for identifying all the possible attacks on a known system. We use a library of templates of the atomic attack steps that can be taken against components in the system and give an algorithm for building these into a tree capturing all the attacks. Future steps will include adapting to other types of networks including wireless and ethernet, and also mixed networks which include networks running under different protocols. We also plan to integrate the automated attack tree generation work presented here into work on model-based security test-case generation which currently assumes the existence of the attack tree such as [3].

References

1. Bryans, J., Nguyen, H., Shaikh, S.: Attack defense trees with sequential conjunction. In: 19th IEEE HASE, pp. 247–252 (2019)
2. Byres, E.J., Franz, M., Miller, D.: The use of attack trees in assessing vulnerabilities in SCADA systems. In: IEEE Conference IISW (2004)
3. Cheah, M., Nguyen, H.N., Bryans, J., Shaikh, S.A.: Formalising systematic security evaluations using attack trees for automotive applications. In: Hancke, G.P., Damiani, E. (eds.) WISTP 2017. LNCS, vol. 10741, pp. 113–129. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93524-9_7
4. Fraile, M., Ford, M., Gadyatskaya, O., Kumar, R., Stoelinga, M., Trujillo-Rasua, R.: Using attack-defense trees to analyze threats and countermeasures in an ATM: a case study. In: Horkoff, J., Jeusfeld, M.A., Persson, A. (eds.) PoEM 2016. LNBIP, vol. 267, pp. 326–334. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48393-1_24
5. Gadyatskaya, O., Jhawar, R., Mauw, S., Trujillo-Rasua, R., Willemse, T.A.C.: Refinement-aware generation of attack trees. In: Livraga, G., Mitchell, C. (eds.) STM 2017. LNCS, vol. 10547, pp. 164–179. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68063-7_11
6. Ivanova, M.G., Probst, C.W., Hansen, R.R., Kammüller, F.: Transforming graphical system models to graphical attack models. In: Mauw, S., Kordy, B., Jajodia, S. (eds.) GraMSec 2015. LNCS, vol. 9390, pp. 82–96. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29968-6_6
7. Jhawar, R., Kordy, B., Mauw, S., Radomirović, S., Trujillo-Rasua, R.: Attack trees with sequential conjunction. In: Federrath, H., Gollmann, D. (eds.) SEC 2015. IAICT, vol. 455, pp. 339–353. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18467-8_23
8. Jhawar, R., Lounis, K., Mauw, S., Ramírez-Cruz, Y.: Semi-automatically augmenting attack trees using an annotated attack tree library. In: Katsikas, S.K., Alcaraz, C. (eds.) STM 2018. LNCS, vol. 11091, pp. 85–101. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01141-3_6

9. Mauw, S., Oostdijk, M.: Foundations of attack trees. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 186–198. Springer, Heidelberg (2006). https://doi.org/10.1007/11734727_17
10. Pinchinat, S., Acher, M., Vojtisek, D.: ATSyRa: an integrated environment for synthesizing attack trees. In: Mauw, S., Kordy, B., Jajodia, S. (eds.) GraMSec 2015. LNCS, vol. 9390, pp. 97–101. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29968-6_7
11. Ray, I., Poolsapassit, N.: Using attack trees to identify malicious attacks from authorized insiders. In: di Vimercati, S.C., Syverson, P., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 231–246. Springer, Heidelberg (2005). https://doi.org/10.1007/11555827_14
12. Schneier, B.: Attack trees. Dr Dobbs J. (1999)
13. Vigo, R., Nielson, F., Nielson, H.R.: Automated generation of attack trees. In: 2014 IEEE 27th Computer Security Foundations Symposium, pp. 337–350. IEEE (2014)