ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

AUTOMATIZATION OF ATTACK TREES

Relatore:
Chiar.mo Prof.
Stefano Ferretti

Presentata da:
Beatrice Spiga

Correlatore:
Nicola Dragoni

Sessione III
Anno Accademico 2021/22

I would like to thank my parents for the support that they always gave me during these two important years of my life, especially during my months abroad. They never had doubts about the fact that I would accomplish this goal.

I wish to extend my special thanks to my grandparents and my uncles because they have always believed in my journey, since I was a child in primary school until the end my master degree.

I would like to say a special thank you to Fabrizio. He has always been my first supporter, even when I had no motivation to continue he always gave me reasons to keep pushing because he believed in me from the very beginning.

A big thank you also to my university mates for the complicity and for the moments that we spent together. Without them all of the group projects would have been very boring.

Finally, I would like to express my gratitude to my thesis supervisors Stefano F. and Nicola D. for making my thesis abroad possible and for helping me work on the topics I like the most. I will bring this experience with me for a lifetime.

Abstract

Oggigiorno la cybersecurity è più critica che mai. L'uso estensivo di dispositivi elettronici espone i nostri dati sensibili a sempre più minacce e vulnerabilità, e tutto ciò può portare a dei cyber-attacchi. Il problema della protezione dei dati e dei sistemi dalle minacce informatiche non ha una risoluzione banale a causa dell'eterogeneità dei sistemi e dei dispositivi esistenti, che possono richiedere mezzi protettivi molto diversi fra loro. Pertanto, giocano un ruolo centrale la prevenzione ed il rilevamento di potenziali minacce nei vari tipi di sistema esistenti. Lo scopo di questa tesi è quello di sviluppare un tool di analisi automatica dei cosiddetti event log dei sistemi che sono stati attaccati o hackerati. L'obiettivo è quello di ottenere un process tree che rappresenti le azioni dell'attaccante partendo dai log, e di usare successivamente alcune regole di traduzione per ottenere un attack tree del sistema in questione. Quest'ultimo può essere visto come una rappresentazione grafica di tutti i potenziali attacchi. Il lavoro proposto può essere utile come mezzo per identificare quali possano essere le possibili debolezze e vulnerabilità che un attaccante potrebbe sfruttare all'interno di un sistema.

Contents

Chapter 1

Introduction

1.1 The problem

Nowadays the digital and the physical universes are becoming more and more aligned and today's information systems log store enormous amounts of events. Companies, hospitals and a lot of other information systems provide detailed information about the activities that have been executed. We refer to such data as event logs. To be more precise, an event log can be seen as a chronological record of activities of a system that are saved to a file on the system [3]. This type of file is useful for administrators because they can monitor how users and processes behave within the system. Moreover, the ability to reconstruct previous operations makes these logs interesting. Indeed, what makes them such a valuable and powerful resource is not only that they give us a lot of information, but they also provide additional data recorded automatically and independently of the person who made the action. To give a more precise idea, Figure 1.1 depicts an example of event log.

No.	Time	Source	Destination	Protocol	Length	Info
605	22.838664	CadmusCo_f6:e2:59	Broadcast	ARP	60	Who has 10.250.176.203? T
606	22.872101	10.250.176.238	10.250.183.255	BROWSER	243	Host Announcement ADM-PC,
607	22.888514	10.250.176.2	224.0.0.2	HSRP	62	Hello (state Active)
608	22.902613	10.250.180.0	239.255.255.250	SSDP	216	M-SEARCH * HTTP/1.1
609	22.914334	MitacInt_50:8d:f1	Broadcast	ARP	60	Who has 10.250.181.10? Te
610	23.080466	10.250.176.2	224.0.0.2	HSRP	62	Hello (state Active)
611	23.096641	10.110.32.66	10.250.180.106	UDP	62	1935→52148 Len=20
612	23.096858	10.250.180.106	10.110.32.66	RTCP	62	Application specific (

Figure 1.1: Example of event log.

In this context, monitoring the system for security reasons is a crucial part for its maintenance, and the generated logs are the basis for discovering security breaches and other issues. All the systems can produce log entries, and identifying malicious behaviors can be very difficult. The larger the number of incoming logs, the more difficult and expensive it becomes to perform manual analysis. Moreover, such data stored is usually stored by information systems in an unstructured form or needs to be labeled with annotations which indicate whether a log corresponds to malicious activity or not. However, the process can be both challenging and costly. This has given rise to a demand for more automated techniques to identify potential system threats. The notion of an Attack Tree is crucial in this situation. For the sake of clarity, an example is depicted below in Figure 1.2.

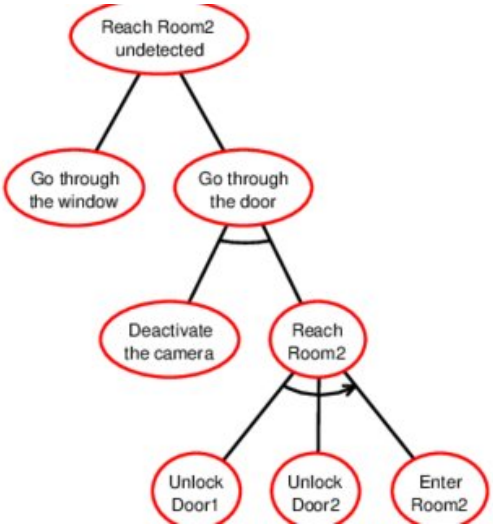

Figure 1.2: Example of Attack Tree.

Attack Trees [1] provide a systematic way of describing the vulnerability of a system, taking into account various types of attacks. Two main aspects make them strong: they combine intuitive representations of possible attacks with formal mathematical methods of analyzing them qualitatively and quantitatively. ([4], [31]). Up to now, analysts and technicians usually constructed Attack Trees manually, based on their knowledge and experience. A large number of tools for editing and analyzing Attack Trees

exist but, unfortunately, their manual design is time-consuming and error-prone. Thus, the resulting trees may be incomplete, i.e., may miss some relevant attack vectors. This can happen especially if their become substantial and security experts may also run into trouble as soon as the material they work on gets fairly big (lengthy log files, for example). Moreover, 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 in the attack vectors that they capture. In addition, a manual design is likely to be incomplete and unsound w.r.t. the security issues of a system under consideration. Supported by automation, practitioners can obtain large Attack Trees that are correct by construction and in line with the properties of the system. The generation process can also be reiterated in case new kinds of attacks emerge or if the system evolves. Even though starting from existing attack patterns may provide valuable help in the design process, 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 recently attract the attention of researchers ([35], [36]).
In this thesis we introduce a tool that allows one to use various transformations to achieve a final goal, which is the automatic generation of an Attack Tree, taking into account the system's event logs. As a result of this operation, it is possible to extract knowledge from violated logs and discover the hacker's steps. In the first step, a process discovery algorithm is used to derive a Process Tree representing the data and operations collected by the system from its log files. To be clearer, an instance of Process Tree is depicted in Figure 1.3. The Process Tree is then translated into an Attack Tree according to specific rules. The aim is to automate the entire Attack Tree generation process, starting from the logs. Due to this work, vulnerabilities that were not considered during the design phase can now be easily identified and mitigated. Moreover, we can have the same perspective of the attacker: this allow one to anticipate and prevent the potential attacks.

Furthermore, this tool can also be used as a forensic tool for investigation, since investigators can analyze the logs of an attacked system a posteriori, extracting information and drawing conclusions based on them.

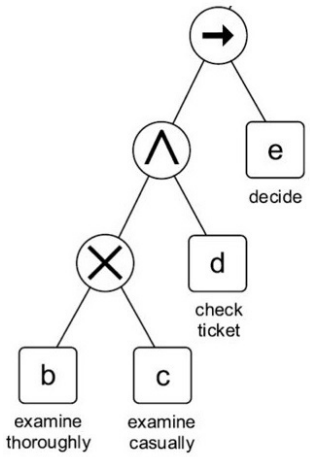

Figure 1.3: Example of Process Tree.

1.2 Contribution of the thesis

In this thesis, we introduce a method to automatically generate Attack Trees by processing event logs, giving the first step towards building an adequate tool for the complete automatization of Attack Trees. Figure 1.4 depicts all the atomic steps. As a starting point, we have three different possibilities. We may immediately start processing some system-provided labeled event logs. Alternately, we can proceed with the generation of some 'dummy' event logs. As a final option, we can employ a model checking tool called Open Fixed Model Checker (OFMC), which accepts protocol specifications as input and outputs attack traces. These attack traces can be finally translated into labeled event logs. As a next step, the Process Tree of the system represented by the logs can be obtained through the use of a Process Mining algorithm. Finally, the system's final Attack Tree can be generated using some translation rules. This will be provided in two formats, one of which is suitable with the security framework RisQFLan that we will examine later..

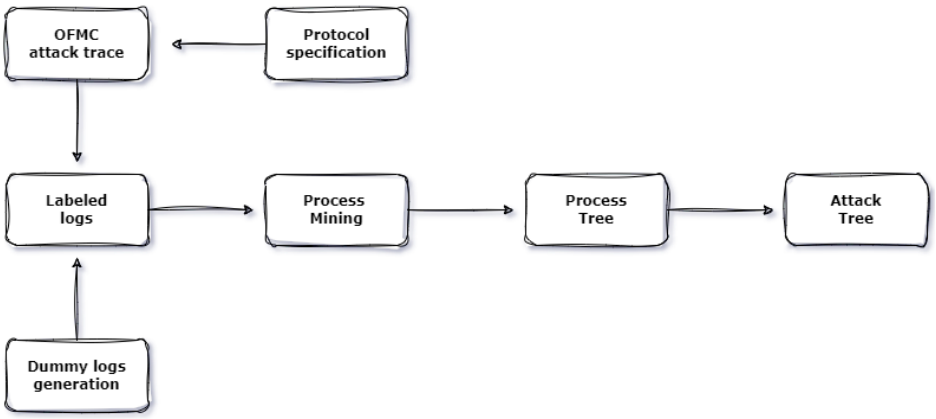

Figure 1.4: Transformations overview.

In particular, the contributions of the thesis are:

- Automatic conversion of OFMC's attack traces, to obtain labeled event logs that can represent the protocol specified as input to the model checking tool. The procedure is introduced in the design part in Section 4.

- Generation of 'synthetic logs', which can serve as a good example to understand the general functioning of the tool. Also this process is outlined in Section 4

- Use of a Process Mining algorithm to build a Process Tree from labeled event logs, which can provide a graphical representation of the system's history. The Inductive Miner algorithm is presented in Section 2

- Use of translation rules on the obtained Process Tree to automatically generate the final Attack Tree of the system. The conversion rules are deeply analyzed in Section 4

- Conversion of the Attack Tree into the format suitable for the graph-based security framework RisQFLan. This enables one to have a different perspective of the same tree, to better understand the overall situation. Also this process is introduced in Section 4.

- Validation of the translation steps through the generation of traces starting from Process Trees, to ensure the validity of the work. This allows one to determine whether the traces are correctly convertible to the translated model and vice-versa. The validation steps are explained in Section 6.

As a result, this framework may enhance the security of many systems, allowing security analysts to protect their assets by choosing proper countermeasures.

1.3 Thesis structure

The thesis is structured as follows. In the next Section the Process Mining technique is described so that we can understand how to obtain the Process Tree of the anaylzed system. Afterwards, Section 3 introduces Attack Trees which are the final result that we want to obtain. In Section 4 the design and the functionalities of our tool are described using a simple example, to better understand the operations involved. Moreover, the OFMC model checking tool is introduced. Subsequently, Section 5 provides analysis of some case studies. In Section 6 the soundness of the obtained model is proven, so that the tool can be considered as validated. Finally, Section 7 draws some conclusions about the presented thesis and considers future work that could improve the tool. Figure 1.5 depicts an overview of the thesis's structure.

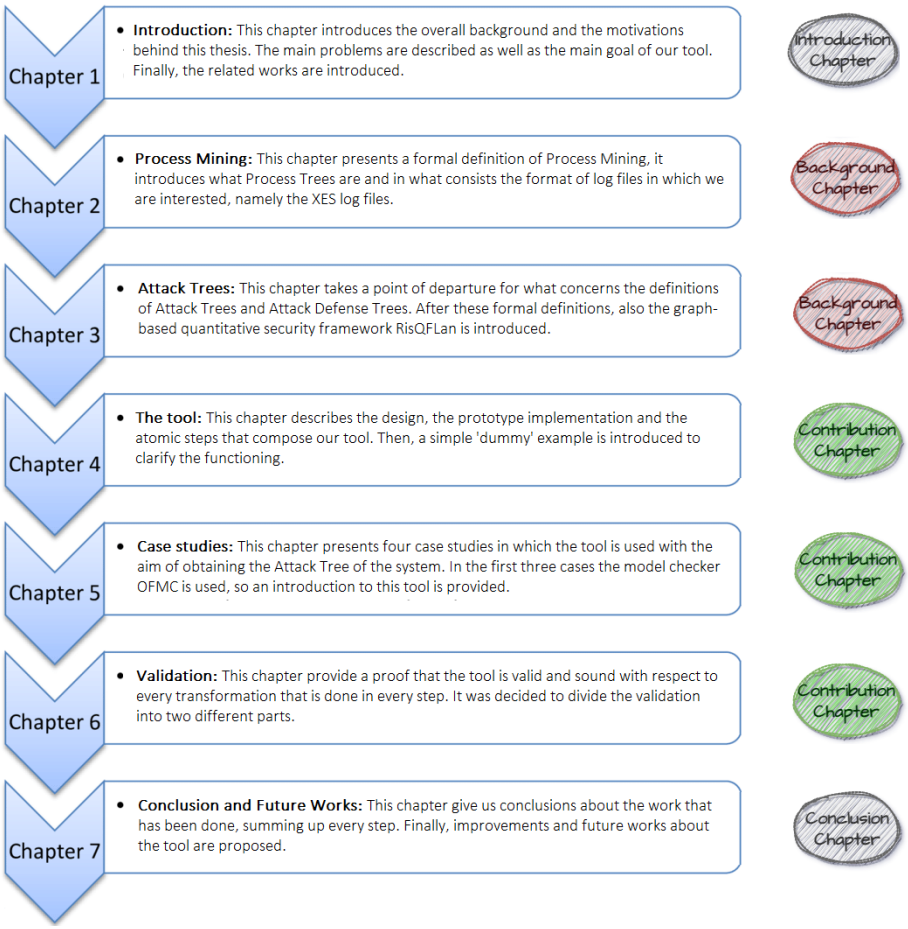

Figure 1.5: Overview of the thesis structure.

1.4 Related works

Attack Trees are broadly used to depict threat scenarios in a succinct and intuitive manner. They are also useful for providing security information to non-experts. Nevertheless, errors can happen because their manual construction relies on the experience and creativity of specialists. This means that manual creation of Attack Trees is impracticable for large system. However, their automated generation has not been fully investigated yet. More specific and detailed related works about Attack Trees, Process Trees, and other analyzed tools, like OFMC and RisQFLan, are mentioned in the related chapters of this thesis. We want to emphasize that, in addition to working on the thesis, we performed research to produce a sur-

vey and illustrate the current the status of automatic generation of two different types of graphical security models, i.e. Attack Trees and Attack Graphs. The paper's purpose is to outline the existing approaches being used in the field, compare them, and discuss the challenges and possible directions for the research community's future. The work is included in this thesis' appendix. We also point out that the survey is still under submission and should be published forthcoming. Hereinafter some existing related works about automatic generation of Attack Trees are proposed.

Phillips et. al. [22] presents a system that generates the graph through the attack profile, the input templates, and the system setup. The technique is built backwards, beginning from the goal node. Next, looking through the templates, it is possible to identify an edge whose head corresponds to the goal node. The routes that do not fit the profile of the attacker are cut out. The procedure continues until an initial state that is not the head of an edge is reached. The authors assert that by changing the network design, their method may model dynamic characteristics. In addition to the network setting and the attacker's profile, a database of typical attacks is also necessary as a precondition. Moreover, code, experiments, proofs, or references to scalability are not available. Because of this, the tool developed in this thesis could contribute and complement the existing works, overcoming these obstacles.

Lallie et al. [23] proposes an analysis of effectiveness in terms of graphical representation of Attack Trees. The quantitative analysis of the visual syntax of Attack Trees is the main contribution of this study. This work examines how these visual structures represent actual cyber-attacks on a system. As a conclusion, it is stated that there is no standard methodology for representing Attack Trees, and that this issue should be fixed by researchers. Although there are many advantages to representing cyber-attacks as Attack Trees, the paper demonstrates that there are inconsistencies regarding the way cyber-attacks are represented in the tree, and by doing so, outlines the need to standardise their visual syntax. This article is the first to provide a thorough critique of attack modeling approaches' visual grammar. Despite this work constitute a good contribution, the au-

tomatic creation of Attack Trees is still not really taken into account. For this reason we decided to create a tool that allows their automatic generation.

Wojciech et al. [24] reviews recent developments in graphical security modeling, with a particular emphasis on the use of formal methods for the interpretation, (semi-)automated generation, quantitative analysis of Attack Trees and their expansions. A comprehensive explanation of existing frameworks is provided in the work, along with a comparison of their features and a list of interesting open questions. Essentially, the work provides formal descriptions of Attack Trees and introduces formal methods to generate them (semi-) automatically. This survey's goal is to provide an overview of current methods that combine formal methods and attack tree-based modeling in the three different dimensions, i.e. semantical, generation and quantitative approaches. However, the suggested semi-automatic generation methodologies rely on complex and sophisticated formal engines ignoring system's event logs, which can represent a rich source of information for the automatic construction of Attack Trees. For these reasons we decided to move on with our tool's design

Kordy et al. [26] examines DAG-based graphical models in the field of security. Direct Acyclic Graphs (DAGs) enable the hierarchical decomposition of complex scenarios into straightforward, intelligible, and quantifiable actions. Two well-known techniques for security modeling are those based on threat trees and Bayesian networks. However, there are more than 30 DAG-based techniques available, each with unique features and objectives. This work aims at providing an overview of DAG-based graphical attack and defense modeling tools. This entails outlining the techniques that are now in use, contrasting their features, and proposing a taxonomy for the formalisms that have been presented. The article also encourages choosing a suitable modeling approach based on user's needs. Essentially, it sums up the current state of available methodologies, to make comparisons and classify them on the basis of formalism. Also in this work there is not a strong focus on the automatic generation of Attack Trees. Indeed, the focus is only on the existing DAG-based graphical

models. Because of this, we needed a tool that produces Attack Trees automatically.

Sheyner et al. [27] uses Model Checking for the automatic construction of Attack Trees or Graphs. The authors here depicts the network as a finite state machine. To automatically produce the tree, the model checker NuSVM has been changed, creating a security property first. The property and the model M (the network) are the inputs for the Model Checker, which returns true if the property is satisfied; otherwise, it gives a counterexample. This counterexample shows a route an attacker could take to go against the security property. The aim is creating Attack Trees for a broader range of features in the future, apart from security properties. In this study, the network settings and the attacker's actions are represented as a set of booleans that express the model's state as state transitions. As a result, the state-space generated is exponential in proportion to the system's variable count. Attack Graphs with an emphasis on network-defined attacks are employed in this work. Although the authors provide experiments and results, they exclude the code and the scalability of their system.

Pinchinat et al. [32] introduced ATSyRA, which is a user-friendly Eclipse-based framework that can help in designing Attack Trees. To secure military buildings, the security expert must first define the system, including the description of the building, the strength of the attacker, and the goal of their attack. The generation of the assault scenarios must be run as the second phase. The input specification is assembled into an attack graph in the current step. The security expert describes a set of high level actions as the third phase (HLA). How it can be divided into smaller activities is implied by the HLA. The run of the Attack Tree synthesis is the last step. The tree in this phase is constructed using the data from Step 3 and the graph created in Step 2. A model checking algorithm is the fundamental interface for achieving this. However, a negative aspect is that the work omits discussing experiments, proofs, and scaling. Additionally, they claim that a fully automated process can produce results that are not sound. We point out that the solution is not entirely automated. Indeed, the security analyst

is actively participating in the process. Consequently, a fully autonomous tool that produces sound results is becoming more and more necessary.

Ibrahim et. al. [33] presents a method that comprises Hybrid Attack Graphs (HAG), which show how an attacked system's logical and actual parameter values can change, as well as possible remedial measures. Automatically created HAGs can be seen using a Java-based tools. In order to complete this operation, the model of the system and the security attribute must be formally described in the AADL language and verified by the JKind model checker. This work provides evidence for how Automatic HAGs can influence the parameters of the particular system being attacked. In order to create this graph starting from some given information about an investigated system and a network, the authors essentially propose a binary classification problem and a multi-output learning technique. However, the work focuses on Hybrid Attack Graphs which have different characteristics from Attack Trees.

David et. al. [34] describes a way for handling socio-technical systems and attacks using timed automata and automated model checking. This type of systems is interrelated with human behaviour. Also simulations and analysis are possible. Model checking is incorporated into this process so that we can identify assaults on the system under analysis. Finally, a detailed example illustrating the use of this approach is given. Nevertheless, the approach is only limited to attacks and system that handle human actions, while we are interested in a more generic and inclusive approach.

We take these surveys as a starting point and, keeping them into account, we introduce the work done in this thesis, which aims to contribute to the existing methods for the automatic generation of Attack Trees. Figure 1.6 summarizes the related papers and their main weaknesses, to better understand the motivation behind our work.

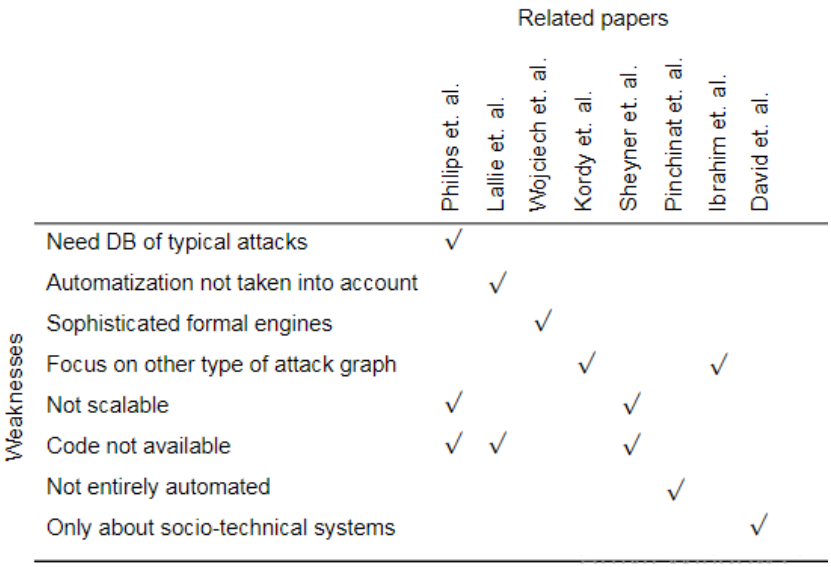

Figure 1.6: Related papers and weaknesses.

Chapter 2

Process mining

In this section, two concepts related to process mining are introduced, i.e. Process Trees and event logs. Basically, they are needed to feed the involved process mining algorithm. These elements make up the first transformation carried out by our tool. Understanding the ideas presented in this section will enable one to fully comprehend chapters 4 and 5, which discuss design and case studies taken under consideration for the thesis.

2.1 XES event logs

Defining the kind of event log required as input for this process is necessary before defining what Process Mining is. For this work we focused on logs in XES format. XES is the successor of MXML (Mining eXtensible Markup Language), the former standard for storing and exchanging event logs [3]. Based on many practical experiences with MXML, the XES format has been made less restrictive and truly extendible. In 2010 the format was adopted by the IEEE Task Force on Process Mining and became the de facto exchange format for process mining.
A XES document contains one log consisting of any number of traces. Each trace describes a sequential list of events corresponding to a particular case. The log, its traces, and its events may have any number of attributes. Moreover, they may be nested. There are five core types: String, Date, Int, Float, and Boolean. XES does not prescribe a fixed set of mandatory at-

tributes for each element: an event can have any number of attributes. However, to provide semantics to such attributes, the log refers to so-called extensions. An extension gives semantics to particular attributes. For example, the Time extension can define a timestamp attribute of type `xs:dateTime`. A log declares the set of extensions to be used. Each extension may define attributes that are considered to be standard when the extension is used.

XES may declare particular attributes to be mandatory. For example, it can be specified that any trace should have a name or each event should include a timestamp. For these reasons, a log holds two lists of global attributes: one for the traces and one for the events. The example log in Figure 2.1 specifies two lists of global attributes. Traces have one global attribute: attribute `concept:name` is mandatory for all traces. Events have three global attributes: attributes `time:timestamp`, `concept:name` and `org:resource` are mandatory for all events.

```
<?xml version="1.0" encoding="UTF-8" ?>
   <extension name="Concept" prefix="concept" uri="http://.../concept.xesext"/>
   <extension name="Time" prefix="time" uri="http://.../time.xesext"/>
   <extension name="Organizational" prefix="org" uri="http://.../org.xesext"/>
   <global scope="trace">
      <string key="concept:name" value="name"/>
   </global>
   <global scope="event">
      <date key="time:timestamp" value="2010-12-17T20:01:02.229+02:00"/>
      <string key="concept:name" value="name"/>
      <string key="org:resource" value="resource"/>
   </global>
   <classifier name="Activity" keys="concept:name"/>
   <classifier name="Resource" keys="org:resource"/>
   <classifier name="Both" keys="concept:name org:resource"/>
   <trace>
      <string key="concept:name" value="1"/>
      <event>
         <string key="concept:name" value="register request"/>
         <string key="org:resource" value="Pete"/>
         <date key="time:timestamp" value="2010-12-30T11:02:00.000+01:00"/>
         <string key="Event_ID" value="35654423"/>
         <string key="Costs" value="50"/>
      </event>
      <event>
         <string key="concept:name" value="examine thoroughly"/>
         <string key="org:resource" value="Sue"/>
         <date key="time:timestamp" value="2010-12-31T10:06:00.000+01:00"/>
         <string key="Event_ID" value="35654424"/>
         <string key="Costs" value="400"/>
      </event>
      ...
   </trace>
   ...
</log>
```

Figure 2.1: Example of a log in XES format.

A subset of the standard attributes defined by the aforementioned ex-

tensions is provided in the list below:

- The concept extension defines the name attribute for traces and events. Note that the example XES file in Figure 2.1 uses `concept:name` attributes for traces and events. For traces, the attribute typically represents some identifier for the case, while for events the attribute typically represents the activity name.
- The time extension defines the timestamp attribute for events. Since such a timestamp is of type `xs:dateTime`, both date and time are recorded.
- The life-cycle extension defines the transition attribute for events.
- The organizational extension defines three standard attributes for events: resource (resource that triggered or executed the event), role and group (they characterize the required capabilities of the resource and its position in the organization).

Moreover, XES logs are supported by a large variety of tools. Therefore, this type of file serves as a view of the event data. The availability of high-quality event logs is essential for process mining. Furthermore, good quality event logs can serve many other purposes. High-quality logs that cannot be tampered with make sure that 'history cannot be rewritten or obscured', and serve as a solid basis for process improvement and auditing.

2.2 Introduction to process mining

The variety of notations available today for process modeling illustrates its relevance. Some organizations may use only informal process models to structure discussions and to document procedures. At the same time, others operating at a higher maturity level may use models that can be analyzed and used to enact operational processes. Today, most process models are made by hand and are not based on a rigorous analysis

of existing process data. There exists a multitude of problems that companies face when making models by hand. Only experienced designers and analysts can create models that have good predictive value and can be used as a starting point for implementation or redesign. An inadequate model can lead to wrong conclusions. Therefore, it's more appropriate to use the event data. Process mining enables the extraction of models based on facts. Moreover, it does not aim to create a single model of the process. Instead, it provides several views of the same reality at various abstraction levels. This is useful also because information systems are becoming more and more intertwined with the operational processes they support. In addition, nowadays multitudes of events are recorded by today's information systems and stored as log files. A log can be seen on various levels, ranging from the meta-level of stored information of the information system itself, to a less abstract level of stored information about what occured in the analyzed device. Regardless of the level we are looking at, the log keeps track of everything that happened in a system and, most importantly, it contains contextual data that is beyond the control of the person who performed the operations. Nevertheless, organizations have problems extracting value from this data. The goal of Process Mining is to use event logs to extract process-related information, e.g., to automatically discover a process model by observing events recorded by an enterprise system. Process mining can be considered as the missing link between data science and process science. The procedure starts from event data and then a process model (representing the process collected in the log) is used in various ways. This is a relative young research discipline that sits between machine learning and data mining on the one hand, and process modeling and analysis on the other hand [3]. The idea is to discover, monitor and improve real processes, i.e. not assumed processes, by extracting knowledge from event logs available in today's systems.The data recorded by the information systems can be used to provide a better overview on the actual processes, e.g. deviations can be analyzed and the quality of the models can be improved. Until recently, the information in these event logs was rarely used to analyze the underlying processes. Process Mining

aims to improve this, by providing tools and techniques for discovering process, control, information, organizational, and social structures from event logs. In general, system logs can be used to conduct three types of process mining: process discovery, conformance checking and process enhancement.

For the first type, a discovery technique takes an event log and produces a model without using a-priori information. If the log contains information about resources, it can also be possible to discover resource-related models, e.g., a social network showing how people work together in a company.

The second type is conformance: in this case an existing process model is compared with an event log of the same process. The conformance checking can be useful to check if the reality, as recorded in the logs, is conform to the model and viceversa. This examination will show us whether this rule is followed or not. Moreover, the conformance checking can be helpful to find, locate and clarify deviations, and to measure their level of gravity.

The third type of Process Mining is enhancement. In this case we try to improve or extend an existing process model, using information about the actual process recorded in some logs. Whereas conformance checking measures the alignment between model and reality, the latter tries to extend or change the a-priori model.

Figure 2.2 depicts how Process Mining states connections between the actual processes and their data on the one hand, and process models on the other hand. The Figure refers to the amount of events recorded by information systems as event log.

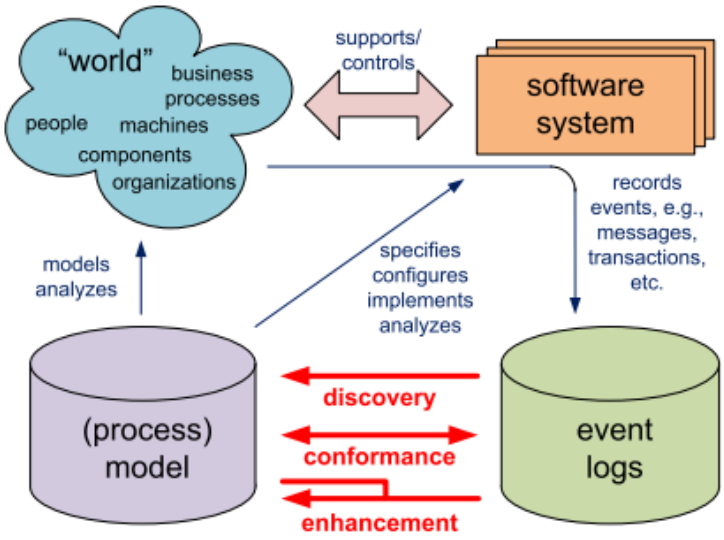

Figure 2.2: Positioning of the three main types of process mining: discovery, conformance, and enhancement.

Focusing on the first process discovery type, i.e. discovery, we aim at mapping the data collected in a XES event log into a so-called Process Tree that explains the behaviors seen in the event log. Several process discovery algorithm exist, such as the Heuristic Miner [38] and the Inductive Miner [40]. Based on the need and the characteristics of the evaluated process, the discovery algorithm is selected and each produces different types of process models.

2.3 Process trees

The vast majority of the most common graph-based process notations, e.g. UML activity diagrams or WF-nets, can be affected by livelocks, deadlocks, and other anomalies. Models having undesirable properties independent of the event log are called unsound. One does not need to look at the event log to see that an unsound model cannot well describe the observed behavior. Process discovery approaches using any of the most common graph-based process notations may produce unsound models. Indeed, the majority of models in the search space are likely to be unsound and this complicates the discovery. It is also possible to use block-

structured models, which are sound by construction. Hereafter, Process Trees (PTs) are introduced as a notation to represent such block-structured models. A Process Tree is a hierarchical process model where the inner nodes are operators such as sequence and choice, and the leaves are activities [3].

Process Trees are tailored towards process discovery. A range of inductive process discovery techniques exists for Process Trees [40], [41]. These approaches benefit from the fact that the representation ensures soundness. A key ingredient of this extremely flexible genetic process mining technique is that the search space is restricted only to sound models.

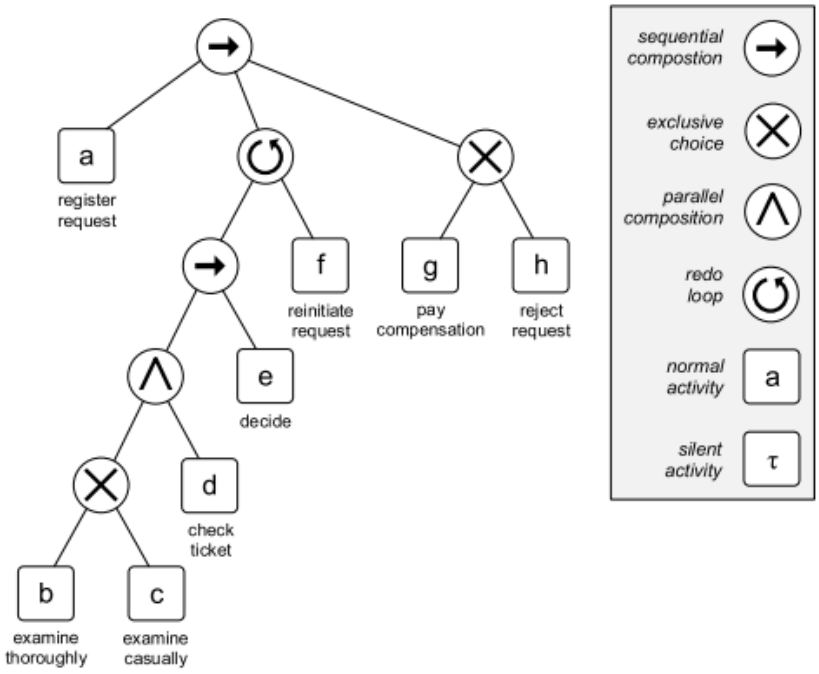

Figure 2.3: Process tree $\rightarrow (a, (\rightarrow (\wedge(\times(b, c), d), e), f), \times(g, h))$ showing the different process tree operators.

Figure 2.3 shows a Process Tree modeling the handling of a request for compensation within an airline. Inner nodes of the tree are operators, while leaves represent activities. There is one root node. The image also shows the four types of operators that are admitted in process trees: $\times$ (exclusive choice), $\rightarrow$ (sequential composition), $\wedge$ (parallel composition), and (redo loop).

The sequence operator executes the children sequentially. The activity a is the first child node of the root in Figure 2.3. Since this is a sequence node, every process instance starts with activity a followed by the subtree starting with the redo loop (). After this subtree in the middle, the execution goes on with the rightmost subtree. The latter subtree models the exclusive choice ($\times$) between g and h. The Process Tree in Figure 2.3 can also be represented in a textual way:

$$\rightarrow (a, (\rightarrow (\wedge(\times(b, c), d), e), f), \times(g, h)) \tag{2.1}$$

The rightmost subtree modeling the choice between the two activies g and h is represented as $\times$(g,h). The redo loop $(\rightarrow (\wedge(\times(b, c), d), e), f)$ starts with the leftmost child and can loop back to whichever of its other children. In this PT, the loop back operation is possible through the 'redo' activity f. The leftmowst child 'do part' is $\rightarrow (\wedge(\times(b, c), d), e)$, namely a sequence that ends with activity e which is preceded by the subtree $\wedge(\times(b, c), d$ where d is executed in parallel with a choice between the two activities b and c. Finally, subtree $\wedge(\times(b, c), d)$ has four possible behaviors: $\langle b, d\rangle, \langle c, d\rangle, \langle d, b\rangle, \langle d, c\rangle$.
It can happen to find the same activity multiple times in a Process Tree, e.g. the tree $\rightarrow (b, b, b)$ handles a sequence of three identical activities. If we analyze $\wedge(b, b, b)$ and $\rightarrow (b, b, b)$, the behaviors are indistinguishable: they have both one only trace that is possible, namely $\langle b, b, b\rangle$.
τ is the so-called silent activity and is not observable, e.g. the Process Tree $\times(a, \tau)$ can model an activity a that is skippable. Another example is (τ, a), which can model a process that executes a any number of times. The 'do' part is silent, and a is in the 'redo' part. Moreover, it is possible not to execute a. Process tree (a, τ) can model a process which executes a at least once. The part of 'redo' is silent: the process can loop back withouth executing activities. The smallest Process Tree is composed by just one activity. If that is the case, the root is also a leaf and there are no operator nodes.

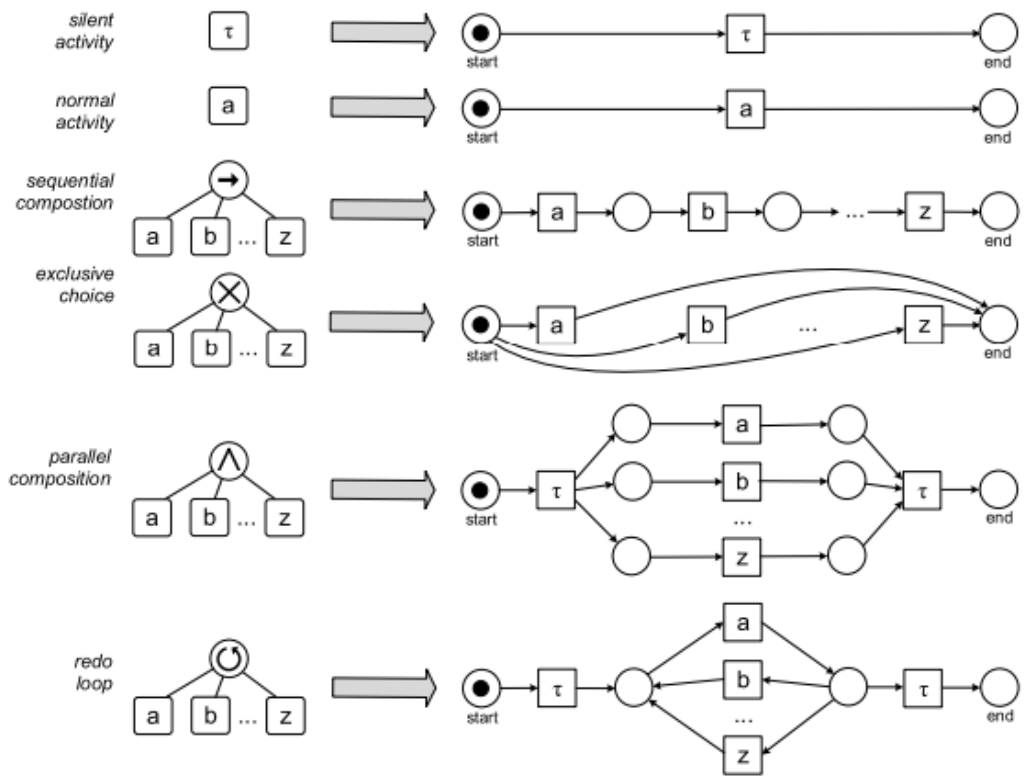

Figure 2.4: Mapping process trees onto WF-nets

In addition, Process Trees allow the conversion in WF-nets (Petri nets) as shown in Figure 2.4. The silent activity τ can be mapped into the transition having the τ as label. The sequential composition $\rightarrow$, the exclusive choice $\times$ and the parallel composition $\wedge$ can be converted in a straight way. In the parallel composition, the τ is used to model the start and the end, to preserve the WF-net structure. The (redo loop) has one 'do' part (activity a in Figure 2.4) and one or more 'redo' parts (activities b until z). In the Petri net, the direction of the arcs indicates the difference in semantics between 'do' and 'redo' parts. The τ is useful to model the entry and exit of a redo loop. This mapping (Figure 2.4) can be used in a recursive way to turn any kind of process tree into a sound WF network. Moreover, the mapping can also be tailored for other types of representation as YAWL, BPML, UML diagrams, and so on. Due to the structured nature of Process Trees, the conversion to other notations is fairly linear.

2.3.1 Inductive Miner

The discovery algorithm that we chose to use in this work is called Inductive Miner (IM) [21]. The IM ensures the rediscoverability of the processes and uses a divide-and-conquer approach. The basic idea is about detecting a 'cut' in the log, e.g. sequential cut, parallel cut, concurrent cut or loop cut, and then recur on sublogs, which were found applying the cut, until a base case is found. Basically, the algorithm decomposes the event log in smaller sublogs, in order to construct a Process Tree. The Inductive Miner ensures that it can re-discover the process model from an event log, since it relies on the directly follows relation between all pair of activities in the event log [3]. The Directly-Follows variant avoids the recursion on the sublogs but uses the Directly Follows graph. Inductive Miner models usually make extensive use of hidden transitions, especially for skipping or looping on a portion on the model. Furthermore, each visible transition has a unique label: there are no transitions in the model that share the same label. The IM algorithm returns a Process Tree, and for this reason we decided to consider it as a valid solution to derive the final Attack Tree at the end of the whole process. The results returned by these techniques can easily be converted to other notations, ranging from Petri nets and BPMN models to process calculi and statecharts. Inductive mining is currently one of the leading process discovery approaches due to its flexibility, formal guarantees and scalability.

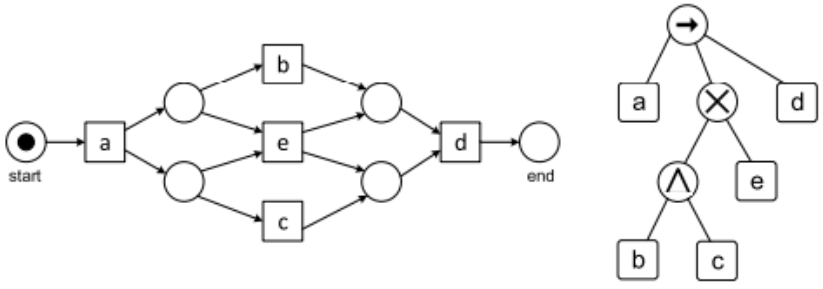

Figure 2.5: WF-net (Petri net) N_1 (left) and process tree Q_1 (right) discovered for $L_1 = [\langle a, b, c, d\rangle^3, \langle a, b, c, d\rangle^2, \langle a, e, d\rangle]$

To be clearer, we take as an example the event log $L \in B(\text{A*})$. This

represents a multi-set of traces over some set of activities A. Our aim is the discovery of a Process Tree $Q \in l_A$. Consider the following event log $L_1 = [\langle a, b, c, d\rangle^3, \langle a, b, c, d\rangle^2, \langle a, e, d\rangle]$ consisting of 6 cases and 23 events. The α-algorithm creates the WF-net $N_1 = \alpha(L_1)$ shown in Figure 2.5 (left-hand side). The basic Inductive Miner (IM) will produce the equivalent Process Tree $Q_1 = IM(L_1) \rightarrow (a, x(\wedge(b, c), e), d)$ also shown in Figure 2.5 (right-hand side). The Process Tree can be automatically converted into the WF-net produced by the α-algorithm using the conversion shown in Figure 2.4, followed by a reduction removing superfluous silent transitions. Any Process Tree can be converted into an equivalent WF-net, BPMN model, etc. Moreover, the basic Inductive Miner (IM) can discover a much wider class of processes and learn "correct" process models in situations where the α-algorithm and many others fail. For clarity we assume that there are no duplicate or silent activities, i.e., in the Process Trees used to generate the example log we will not find two leaves with the same activity label and leaves with a τ label (silent activity).
Given that we just described what a Process Tree is, we can see below in what Attack Trees consist. By doing so, the translation process between these two types of graphs can be better understood.

Chapter 3

Attack trees

This chapter introduces the concept of Attack Tree, compares it with Attack-Defense Tree, and describes the RisQFLan security framework employed in the last part of the thesis. By understanding these concepts, one can understand the final transformations that our tool performs. Chapters 4 and 5, which are about design and analyzed case studies, will make use of the notions introduced in this Section.

3.1 The definition

The term Attack Tree was first introduced by Schneier in [1]. ATs form a convenient mean to systematically categorize the different ways in which a system can be attacked. An Attack Tree is a graphical tree-structured representation of the system's security showing possible attacks. The main idea is to decompose a given task into smaller ones, making easier to describe and quantify different metrics. Since Schneier introduced Attack Trees, multiple approaches and formal semantics have been proposed in the literature. This thesis will focus primarily on [4], since it presents the structure and foundations of this tree in a formalized form, based on Schneier's model.
The purpose of an Attack Tree is to determine and examine all the possible attacks against a system in a structured way. This specific structure is expressed in the hierarchy of the nodes that compose the tree, allowing the

decomposition of an attack goal into a certain number of concrete attacks or sub-goals. This structure brings us information about the interpretation and grouping of attacks. Attack trees provide us with an easy way to define a collection of possible attacks. They can also be called attack suite. These attacks consist in components required to perform the attack. A certain component may occur more than once, so an attack can be seen as a multi-set of attack components. As a result, an attack is considered to be at the lowest level of abstraction, and it does not have an internal structure. A simple example of Attack Tree is shown in Figure 3.1.

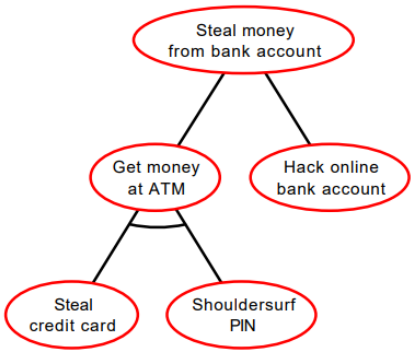

Figure 3.1: An attack tree representing stealing money from someone's bank account.

In Attack Trees the nodes represent attacks, while the root is the global goal of the attacker. Children of a node are refinements of this goal, and therefore leafs represent attacks that can no longer be refined. The structure of the tree highlights the vulnerabilities of the system and helps developers and analysts to focus on weak spots, so that countermeasures may be implemented. With Attack Trees one can form multiple attacks derived from physical, technical or human vulnerabilities. A refinement in the tree can be conjunctive (aggregation) or disjunctive (choice). If the arc connection of two components is a conjuntive refinement, then all subgoals have to be fullfilled, and the node is called an AND node. On the other hand, disjuntive ones express that satisfying one sub-goal is enough to achieve the parent goal, and these nodes are called OR nodes [4]. In addition, another considered refinement is SEQUENTIAL-AND (SAND), in

which all the subgoals must be fulfilled to reach the parent goal, but with the additional constraint that the sub-goals must be carried out in a given order [37].
As an example, we can observe in Figure 3.1 that the root node represents the main goal of the attack, i.e. to steal money from a bank account. The attacker's aim is disjunctively refined into two alternative sub-attacks, namely the attacker may try to get money from an automated teller machine (ATM), or he might attempt to hack the online bank account system. The sub-goal that explores getting money at an ATM is further conjunctively refined into two complementary activities: the attacker must steal the credit card of the victim and he also needs to obtain the PIN code by shoulder-surfing. Note that a conjunctive refinement is denoted graphically with an arc spanning the child nodes.
Once the possible attacks on a system have been modeled in the Attack Tree, this graph can be used to analyze attributes about system's security. Schneier [1] suggests several such attributes, assigning values on the leaf nodes in order to make calculations about the security of a goal. Some examples could be about the (im)possibility, the cost, and whether special tools are needed. As soon as all the leaf nodes have a value, these amounts are propagated to the root of the tree. The OR value of a node is Possible if one of the sub-goals is Possible. On the other hand, an AND value of a node is Possible if all its sub-nodes are also Possible. In addition, Attack Trees offer the opportunity to calculate different metrics associated with a specific attack. To accomplish this, some attributes can be added to the tree. These attributes could also be used for quantitative analysis on the tree [5].
For the scope of this thesis, the introduction of the concepts formally introduced by Schneier was necessary. Understanding these definitions is a prerequisite for introducing our tool.

3.2 Quantitative security framework

Quantitative modeling and analysis approaches are essential to support software and system engineering in scenarios where qualitative approaches are inappropriate or unfeasible, for example due to complexity or uncertainty. Automated approaches to support quantitative modeling and analysis have been widely developed during the last years, including both generic and domain-specific approaches (cf., e.g., [25]). QFLan [2] is an example of successful domain-specific approach to support quantitative modeling and analysis of configurable systems, like software product lines. It combines various well-known rigorous notions and methods in an Eclipse-based domain-specific tool framework. It consists of a domain-specific language (DSL) tailored for configurable systems, and an analysis engine based on statistical model checking (SMC) [28] [29]. In the next subsection, we will introduce a new framework that can support quantitative security risk modeling and analysis based on attack-defense diagrams, which originates from QFLan.

3.2.1 RisQFLan

In the present thesis, we used a novel tool, called RisQFLan [30], which is a framework that can support graph-based quantitative security risk modeling and analysis. To be precise, by combining distinctive characteristics of existing formalisms, this tool allows one to build rich models for risk modeling and analysis. The DSL of RisQFLan was designed to include the most significant features of existing formalisms based on Attack Trees. In this way it is possible to combine them in the same model. Moreover, this framework allows one to fine tune security scenarios by defining explicit attack behavior, implicitly constrained by an attack-defense diagram. Explicit attack behavior enables the analyses of specific attacker types, like script kiddies, insiders, and hackers. The benefit of this is that we are able to identify vulnerabilities for attacker types that are most appropriate for the security situation at hand. Furthermore, this enables

novel types of analysis that complement the classic best- and worst-case evaluations of attack graphs. The attack behavior is modeled as rated transition systems, whose transitions are labeled with the action being executed and a rate (used to compute the probability of executing the action), and possibly with effects (updates of variables) and guards (conditions on the action's executability).

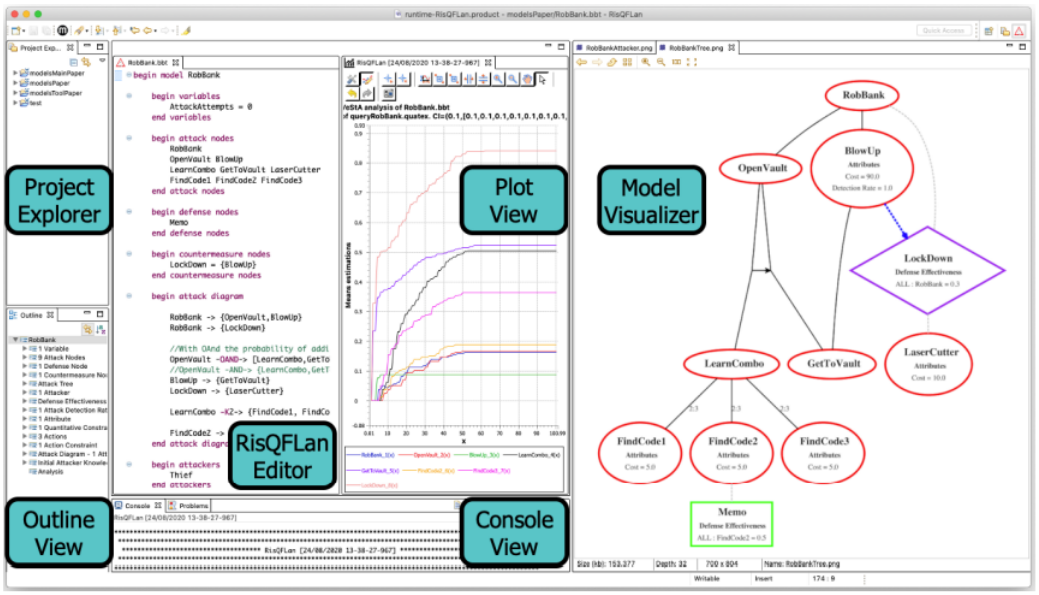

Figure 3.2: The RisQFLan tool.

The tool's architecture is organized in two layers: the Graphical User Interface (GUI), devoted to modeling, and the CORE layer, used for the analysis. Both layers are wrapped in an Eclipse-based tool embedding the third-party statistical analyzer, as we can see in Figure 3.2.

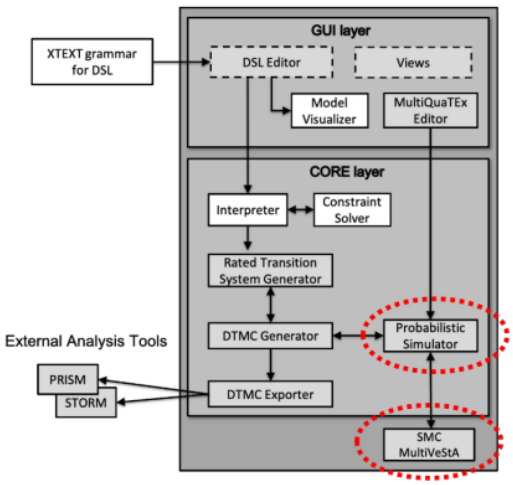

Figure 3.3: The RisQFLan architecture [30].

In Figure 3.3 the RisQFLan architecture is shown [30]. Here the various elements of the framework are depicted. Circles in red represent the probabilistic simulator and SMC: they interact to estimate the models' properties in a statistical way. RisQFLan also enables one to specify a probabilistic attacker behavior. This allows one to make specific analyses on certain typologies of attacker. The option that permits the specification of different attacker types can help to evaluate the vulnerabilities of a system and allocate the right assets needed to protect it. In this framework, transitions are labeled with the executed actions, and the rates represent the probability computation of completing one action. Besides, RisQFLan supports quantitative constraints to permit the imposition of limitations on the attacker's behavior by setting a total cost that he can spend during the offense. All the attack attempts will cost the attacker even if he fails. Until the total cost is not reached, he can keep trying to perform the attack. If the attacker run out of resources, he cannot start any further attacks on the system. These quantitative constraints allow the testing of a system against different types of attackers with various kind of assets. The next subsection will introduce a straightforward visual language to describe an attack scenario that RisQFLan employs, i.e. Attack-Defense Trees.

3.3 Attack-defense trees

We already introduced in Section 3.1 the notion of Attack Trees (AT). They have been introduced by Schneier in 1999 and have been used for many years to analyze potential attack scenarios. A limitation derived from the Attack Trees is that they cannot depict possible countermeasures that someone could take in order to contrast attacks. Taking this limitation into account, Kordy et al. [**?**] proposed a variation of Attack Trees, i.e. Attack Defense Trees (ADT), which also take into account defense mechanisms in the attack context. Indeed, Kordy extended the Attack Tree proposed in [4]. Similar to ATs, Attack-Defense Trees include refinements and countermeasures. Each node can be refined into various sub-goals, and they can also have one child of the opposite type. Child nodes are conditions that must be fulfilled to achieve the goal of the parent. Leaves represent atomic steps that the attacker must take, that cannot be refined. What is different from the Attack Tree model is that the goals of the actors involved can be countered by goals of the adversary, which themselves can be countered again, and so on. Attack–Defense Trees [**?**] are one of the most well-studied extensions of Attack Trees, and every year many new methods are developed for their analysis. They can enhance ATs with nodes labeled with the goals of the defender, enabling the modeling of interactions between two competing actors. Attack-Defense Trees have been used to evaluate the security of real-life systems, like ATMs, cyber-physical systems and RFID managed warehouses. The theoretical developments as well as the practical studies showed that Attack–Defense Trees give us a good method for the security evaluation, but they also highlighted room for improvements.

Both types of tree constitute a popular family of graph-based security models, for which several approaches have been developed over the last years, e.g., [5].Their goal is to provide scalable and usable methods for specifying vulnerabilities and countermeasures, their interplay and their key attributes such as cost and effectiveness. Thus, Attack Trees (and Attack-Defense Trees) provide a basis for quantitative risk assessment. In

this way, it can be determined whether the defensive assets are sufficient to protect a system or whether they must be allocated differently. In Figure 3.4 a refined version for Attack Defense Trees is depicted. This type of ADT includes different nodes that dictate various relations between parent and child nodes. The tree includes AND nodes, in which all the subnodes or leaves need to be activated, i.e., the attacker has completed all the sub-attacks. It also includes OR nodes that can be activated when at least one sub-attack needs to be completed by the attacker. This version of tree also includes defense nodes, such as countermeasure and defense nodes. The former are reactive defenses so that, whenever an attack is detected, the countermeasure is activated. On the other hand, defense nodes are always active on the vulnerability that the attacker wants to exploit. RisQFLan offers also the possibility to model Attack-Defense Trees, and it can handle both defense and countermeasure nodes. Furthermore, the refined version of ADT employed in RisQFLan includes other two types of nodes, namely Ordered-AND and K-Out-of-N nodes. For the former, the attack on the sub-goals should be completed in a particular order to achieve the parent node. On the other hand, for the latter, it is enough for the attacker to complete K out of N sub-goals to activate the parent node.

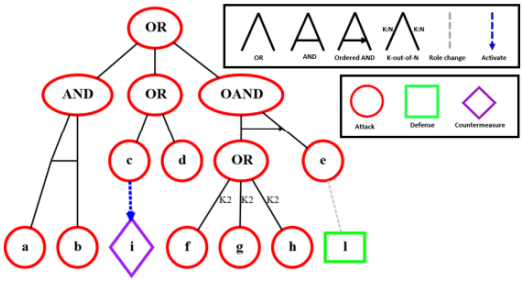

Figure 3.4: Refined Attack Defense Tree.

For the sake of clarity, we take as starting point the Attack Tree about the bank theft of money, previously introduced in Figure 3.1, turning it into the Attack Defense Tree depicted in Figure 3.5. Note that now also defense nodes appear in the graph. In this tree, the proponent is the attacker and the opponent is the defender. According to the Attack–Defense Trees convention that we just introduced, nodes that represent goals of an at-

tacker are depicted using red circles, while those about the defender using green rectangles. Every countermeasure is attached to nodes they should counter via dotted edges. One way to protect against a PIN shouldersurf could be to memorize it, while hacking an online bank account could be prevented by using a second authentication factor. This defense node is refined into two distinct sub-nodes that represent two concrete defenses, i.e., key fobs or PIN pad.

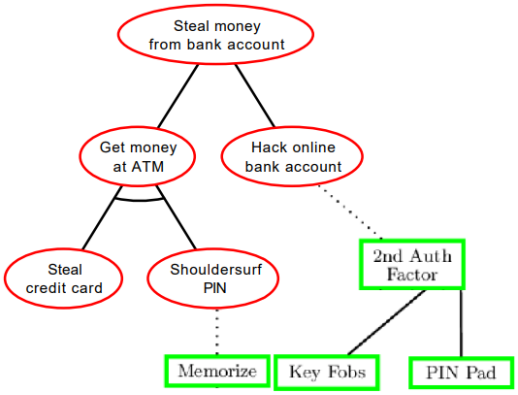

Figure 3.5: An Attack Defense tree representing stealing money from someone's bank account.

The growing variety of methods for quantitative analysis of Attack–Defense Trees provides security experts with various tools for determining the most dangerous attacks and the best ways of securing the system against those threats. Indeed, AD diagrams are a useful tool for modeling and reasoning on contexts about security risks. Standard analyses conducted on Attack-Defense diagrams are usually about the feasibility of attacks, their likelihood or their cost. Techniques of analysis are often based on constraint solving, optimization and statistical techniques. Attack Trees (and Attack-Defense Trees) are broadly used in various domains like, e.g., defense or aerospace.

For the scope of this thesis we will focus only on Attack Trees, also with respect to the use of the RisQFLan tool, but future works could also include Attack-Defense Trees or any other kind of attack diagram that includes countermeasures.

Chapter 4

The tool: from event logs to attack tree

4.1 Design

The main goal of this section is to explain the design of the created tool, which is composed by several transformations, starting with an event log and leading to an Attack Tree. All the translations were depicted also in the introduction section in Figure 1.4, and each will be explained more in depth hereinafter. Subsequently, an example concerning a 'dummy' log generation is provided.

4.1.1 Labeled logs

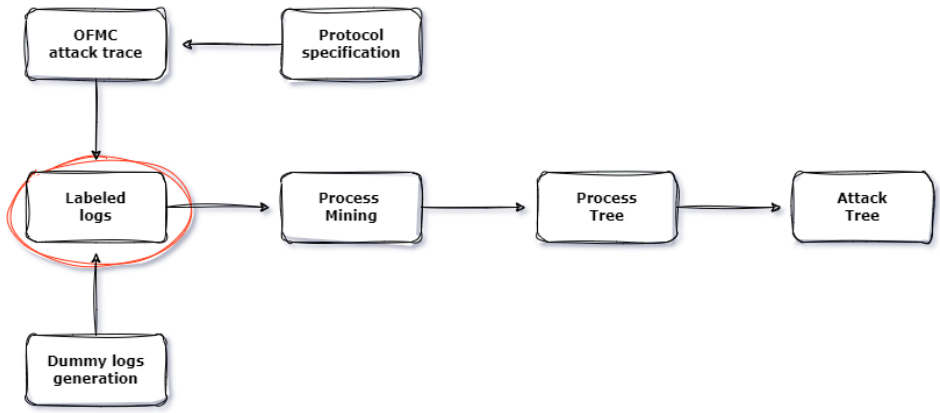

Figure 4.1: Transformations overview: system's event logs.

To begin with, we need an event log related to a specific system as initial input to the tool. An event log is a time-stamped record of a system's actions, recorded into a file on the system itself [3]. The admin can use this file to monitor how users and programs behave within the system. The capacity to reconstruct previous processes makes these records very noteworthy. They supply us with wealth information, but they also provide additional data that is recorded automatically and independently from the individual who performed the action. This is what makes them such a valuable and powerful resource. In our tool, the Process Mining algorithm is fed with these logs, to obtain a Process Tree related to the analyzed system, as we will see later in this chapter. It is also possible to generate some 'dummy' event logs as an alternative input for the tool. To make the Process Mining technique work properly, the system's log must include the following information:

- A case ID that identifies a sequence of simulated attacks.
- An activity name that indicates which action has been executed in each event. To be more precise, we need labeled event logs in order to fill this field.
- A timestamp that identifies the time and date of the performed activities.

- (Optional) Information/content: includes a label about the specific atomic activity.

If real system's event logs are available, the preliminary part about the creation of 'dummy' logs can be skipped. The generation can be done using a Python library, as further introduced in the next section.

4.1.2 Dummy logs generation

In case we do not have real event logs, we can generate some fake ones by creating a Pandas Dataframe containing the following information: ID, activity name (i.e. the label), and timestamp. These are the most essential details that must be present in any event log, regardless of whether they are generated or not, as we previously stated. An in-depth description of the code for the generation can be found in the 'dummy' example at the end of this chapter.

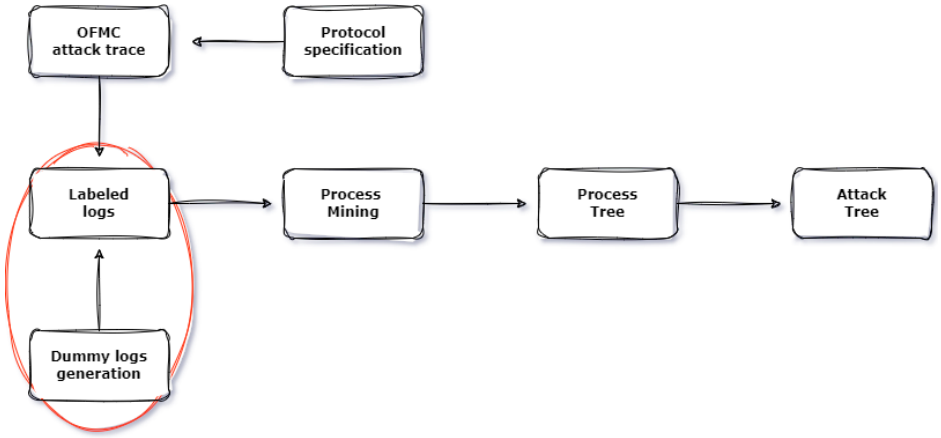

Figure 4.2: Transformations overview: dummy log generation.

Afterwards, we need to convert the obtained Dataframe, exporting it as event log object in XES format. To do this, the Python package PM4PY comes in handy. It uses the most recent, effective, and thoroughly examined Process Mining techniques. The PM4PY function `format_ dataframe` can be used to accomplish this. Then, the converter can transform the Dataframe by default into an Event Log object. The 'dummy' example section that follows will provide a thorough explanation of the code. Now,

as a finishing part, the `write_xes` function can be used to convert the log into a XES file. In this approach, the event log is exported to an external file with the right format, so that the Process Mining algorithm can use it as input data.

4.1.3 Opensource Fixedpoint Model Checker tool

The automated analysis of security protocols is a field that stands between formal methods and IT security, and it has been intensively studied during the last two decades. We introduce now the Open-source Fixedpoint Model Checker OFMC [18] for symbolic security protocol analysis. OFMC performs both falsification of protocols, namely detection of attacks, and session verification by exploring the transition system resulting from an IF specification. OFMC's effectiveness is due to the integration of a various symbolic, constraint-based techniques that are proper and terminating. This tool also permits the analysis of security protocols with respect to an algebraic theory of the employed cryptographic operators, which can be specified as part of the input [39].
The native input language of this tool is the AVISPA Intermediate Format IF [19], [20]. It describes a security protocol as an infinite-state transition system using set-rewriting. However, for the purpose of this thesis we will use AnB, another language that is supported by OFMC. AnB specifications are automatically translated to IF and this translation defines a formal semantics for AnB in terms of IF. This language consists in an Alice-and-Bob-style language that describes how messages are exchanged between honest agents acting in the different protocol roles. Its purpose is to extend previous similar languages supporting algebraic properties of cryptographic operators, with a simple notation for different kinds of channels that can be used as assumptions and as protocol goals. With AnB, channels can be specified as assumptions (when the protocol is based on channels with specific properties for the transmission of some of its messages) and as goals (when a protocol should establish some kind of channel). This novel language can provide support for protocols that require algebraic

properties for the protocol execution, and it can also provide a notion of the different communication channels that can be used to achieve a protocol's goals and assumptions [39].

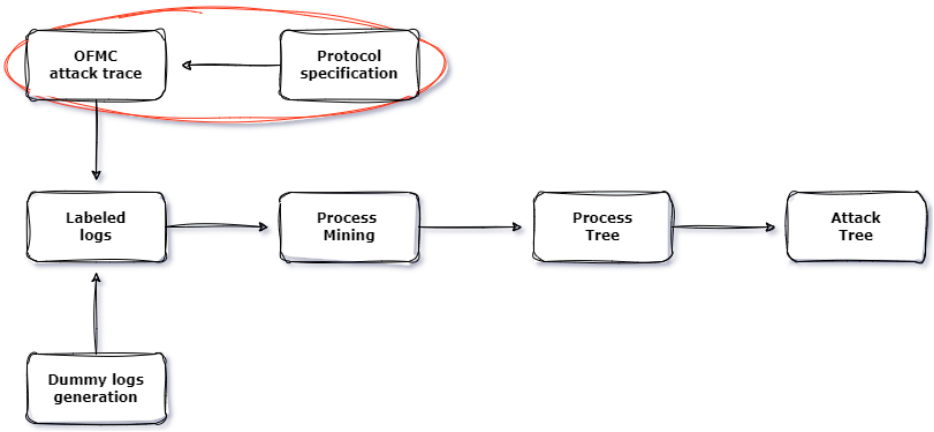

Figure 4.3: Transformations overview: OFMC attack trace conversion into logs.

Proceeding with the description of the design, we can obtain an attack trace involving one or more attacks against a specific protocol after passing that protocol as an input to the OFMC tool. Then, using the Pandas package from Python, we can translate the acquired trace and produce a Dataframe of logs from it. Afterwards, the Dataframe can be exported as a XES log file for Process Mining. As we outlined before, the specific columns of data required in the log are:

- IDs: an identification number of a single activity in the logs, to distinguish it from other events.
- Timestamps of activities: all events have a date and time associated. This is useful when analyzing performance related properties, e.g., the waiting time between two activities. If an event log contains timestamps, then the ordering in a trace should respect these timestamps, i.e. they should be nondescending in the event log.
- Activity description: this field includes the label that describes what this line of log is about, together with the content of the message exchanged between the two parties. The content of the message is

useful to understand why the weak authentication attack is happening and what the intruder is doing. In this specific case, the intruder is faking his real identity: he claims to be x29 with the server, but to the receiver x401 he declarates to be x27. Here the authentication is broken.

- Information/content: this column contains the label to describe the single atomic activity.

To isolate these information, a translation script was written so that the Pandas Dataframe could be generated. The specific script is depicted and analyzed in the next section concerning the 'dummy' log generation example. This is suitable to convert any kind of attack trace generated by the OFMC tool, even the most complex ones.

4.1.4 From labeled event logs to process tree

Continuing, now the Process Mining technique can be applied on our event logs. It is not important if the logs come from an actual system, an OFCM attack trace, or they are generated. We already discussed process mining and process trees in chapter 2, to provide some background information for this portion of the design. Process mining tools can be used to refine the analysis, e.g., by applying filters that help focus on specific properties. This can help identify undesirable behavior in the discovered process model. The Process Tree of the examined system is obtained from the event log as a consequence of Process Mining. This conversion allows process discovery using the Inductive Miner algorithm. Indeed, the Inductive Miner aims at deriving a Process Tree which represent the behaviour listed in the event log.

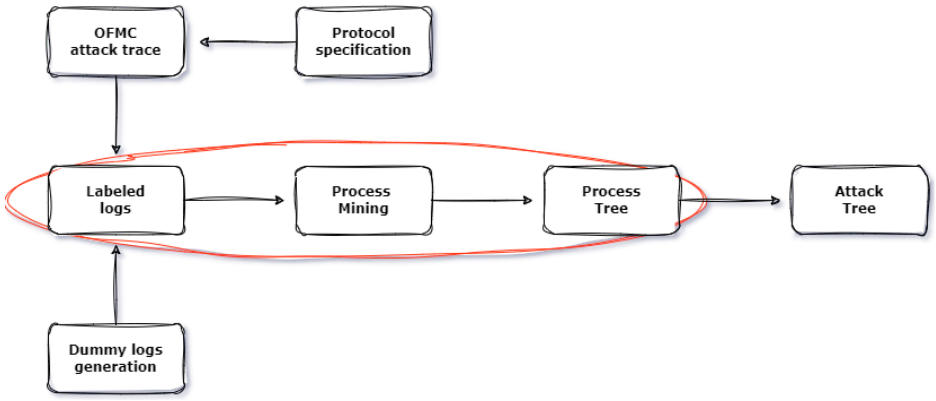

Figure 4.4: Transformations overview: from event logs to Process Tree through Process Mining.

Using the same Python library mentioned before, i.e. PM4PY, we can apply Process Mining to our event logs. An implementation of the Inductive Miner algorithm is provided in this package. Petri Net and Process Tree are the two process models that can be derived utilizing the IM. To do this, a log is read, the Inductive Miner is applied, and a Process Tree (or a Petri Net) along with the initial and final marking are found. The 'dummy' log generation example in the following section will go more into detail about this process.

4.1.5 From process tree to attack tree

The next goal is the conversion of the obtained Process Tree into the related Attack Tree. Also the vice-versa is possible, but for the scope of this thesis we are interested just in the translation process from PT to AT. In chapter 3 we introduced the notions of Attack Tree and Attack-Defense Tree, to better understand the part of the tool that performs this translation. Now we introduce the characteristics of the Process Tree that is mapped into an Attack Tree. Next in this section we also describe the translation rules for the two languages involved in this process [16].
As already mentioned, Process Trees belong to the family of process models. A process model describes the flow of executed activities to accomplish a certain goal. The goal of a process model is to describe activities

and relationships, and their execution order [3]. In a Process Tree, internal vertices are relations, and leaves represent activities. On the other hand Attack Trees, by definition, are composed by a main goal (the root node), subgoals (intermediate nodes), and actions (leaves). The main goal is decomposed into subgoals. Each attack consists of components required to perform the attack itself. The main difference between these languages is that a Process Tree does not directly model the goal, since it is the objective of the model representation itself, while an Attack Tree directly models activities and goals. For the purpose of the translation, we consider the last activity to be executed in the Process tree as the root node of the Attack Tree, i.e. the goal. This is done in order to achieve the attack goal. Firstly, we need to execute the list of activities required and, at the end, we reach the final node. Therefore, the root node in the Attack Tree is replaced with the sequence operator in the Process Tree. Note that the notion of observable and non-observable actions is presented in [16]. This definition is needed since the concept of goal is not embedded in the Process Tree. This diversification is useful because ATs have a goal-oriented and self-explainable structure, while PTs require that each activity node is observable in the event log. For this reason, we distinguish between:

- Observable action: individual action collected by an information system that can contribute to the achievement of the goal. It can be perceived as an atomic event.
- Non-observable action: goal which cannot be directly mapped to an execution of an activity. It can be viewed as a concept.

A non-observable action will be skipped in the Process Tree. As introduced in chapter 3, Attack Trees also involve the conjunction (AND) and disjunction (OR) operators, while Process Trees define four sequential operators (sequential, exclusive choice, parallel composition and redo loop), as described in chapter 2. In order to translate a Process Tree into an Attack Tree, the latter must be able to represent the operators of the former. While the conjunction operator (AND) already finds a definition in the

Process Tree, that is the parallel operator, the disjunction (OR) must be defined. In a Process Tree, the OR operator is a node whose child-nodes can all be executed, or only partially, and the execution order does not matter. Eventually, some operators from the Process Tree are discarded because a correspondence in the Attack Tree is not found, e.g. the redo loop, because for definition Attack Trees are directed acyclic graphs.
Once the translation rules are applied on the Process Tree, the related Attack Tree is generated. The resulting tree is in XML format, but for the scope of this thesis another translation is performed as the last step of the overall process.

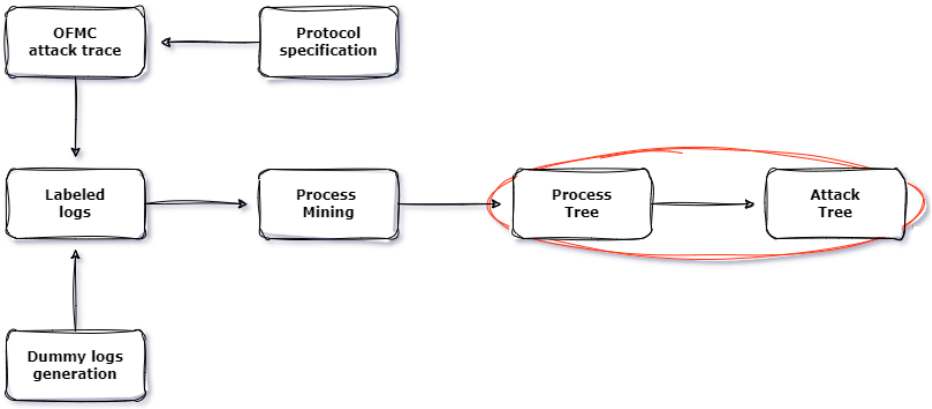

Figure 4.5: Transformations overview: Process Tree to Attack Tree.

Hereinafter, the translation rules from Process Tree to Attack Tree are presented. Note that in this thesis we are interested only in the transformation of a Process Tree into an Attack Tree, and not vice-versa, although the reversed process is also possible [16]. We start analyzing each operator concerning Process Trees.
Figure 4.6 involves the sequence operator that is used to define the parent-child relationship and the rightmost child in a PT: this will become the root node in the AT. Indeed, activities in the Process Tree are executed to reach a final goal, namely the last action, which is the root note in the Attack Tree. The procedure for 4.6 consists in the copy of T1 to all the leaves of T2.

Figure 4.6: From Process Tree to Attack Tree: sequence operator.

For what concerns the AND operator, the translation is done by keeping the same child nodes involved in the process but the relation becomes a conjunction, and the root turns into a non-observable action. The procedure is similar for the OR operator, because it is translated into the disjunction between the same children. These two processes can be viewed in Figure 4.7 and Figure 4.8.

Figure 4.7: From Process Tree to Attack Tree: AND operator.

Figure 4.8: From Process Tree to Attack Tree: OR operator.

To further explain the rules of translation, now we provide an example. In Figure 4.9 the Process Tree contains the sequence operator as root, and the exclusive choice (OR) in the rightmost child. Following the rules, the rightmost child of the sequence operator is translated in root (in this case there are two nodes), and both of them have as child the left part of the sequential composition in the Process Tree. The same case occurs when the Process Tree is composed by the sequence operator and the inclusive

choice (AND), but the latter is translated into a conjunction in the Attack Tree.

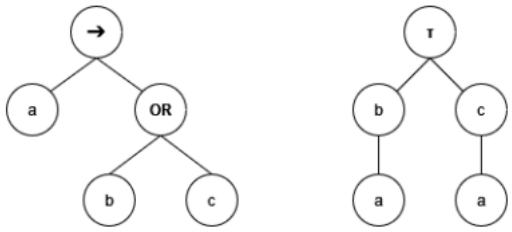

Figure 4.9: Complete example of translation from PT to AT.

4.1.6 Conversion of the attack tree for RisQFLan

To better understand the last part of the tool, an introduction to the RisQFLan tool was provided in chapter 3. This portion of work entails converting the obtained XML Attack Tree into a format compatible with the RisQFLan framework, while maintaining the same version of the tree. This operation allows one to exploit a programming-like environment that is attractive to software developers, and that integrates at the same time a graphical component which make it more appealing for security experts. In RisQFLan's Domain Specific Language (DSL), nodes are defined in specific blocks as we can see in Figure 4.10. Moreover, it is also possible to define defense and countermeasures nodes. This is feasible since the framework also considers Attack-Defense Trees, but for the scope of this thesis we will focus only on Attack Trees. The operators OR, AND and OAND (ordered AND) are offered. AND and OR refinements originate from the seminal works on Attack Trees [1]. OAND refinements stem from enhanced and improved Attack Trees [15] and are used to model ordered attacks: sub-nodes can be activated in any order but only the correct one activates the parent node. Essentially, square brackets of OAND in Figure 4.11 indicate that order matters: Node1 requires Node2 and Node3 in that order.

```
begin attack nodes
    Node1 Node2
    Node3 Node4
    Node5 Node6 Node7
end attack nodes
```

Figure 4.10: Example of attack nodes in the RisQFLan framework.

```
begin attack diagram
    Node1 -OAND-> {Node2, Node3}
    Node2 -AND-> {Node4, Node5}
    Node3 -OR-> {Node6, Node7}
end attack diagram
```

Figure 4.11: Example of attack diagram in the RisQFLan framework.

With the completed procedure, we are able to fully automate the process and produce the examined system's Attack Tree in two different formats, based on the analyzed event logs.

4.2 A dummy example

In this section a straightforward example is provided, to better explain the general procedure. Although we could have used some real event logs, in this instance we decided to show the dummy log generation for the sake of simplicity. We start by creating a Pandas Dataframe containing ID, activity name and timestamp. An example of dummy log generation procedure is shown in Figure 4.12. On the other hand, Figure 4.13 refers to the resulting Dataframe of logs.

```
data = [['0', datetime.now(), "4624 - An account was successfully logged on"],
     ['0', datetime.now()+timedelta(minutes=1),
    "4625 - An account failed to log on"], ['0', datetime.now()+timedelta(
        minutes=2), "4626 - User/Device claims information"], ['0', datetime.
        now()+timedelta(minutes=3), "4627 - Group membership information."],
    ['0', datetime.now()+timedelta(minutes=4), "4634 - An account was logged
        off"], ['1', datetime.now(), "Eavesdropping"], ['1', datetime.now(), "
        Malware"], ['1', datetime.now(), "Exploit IoT vulnerabilities"],
    ['1', datetime.now()+timedelta(minutes=1), "Take control of IoT device"],
        ['1', datetime.now()+timedelta(minutes=2), "Conduce DDoS on multiple
        targets"],
    ['2', datetime.now(), "Malware"], ['2', datetime.now(), "Eavesdropping"],
        ['2', datetime.now(), "Exploit IoT vulnerabilities"],
    ['2', datetime.now()+timedelta(minutes=1), "Take control of IoT device"],
        ['2', datetime.now()+timedelta(minutes=2), "Conduce DDoS on multiple
        targets"],
    ['3', datetime.now(), "Exploit IoT vulnerabilities"], ['3', datetime.now()
        , "Malware"], ['3', datetime.now(), "Eavesdropping"],
    ['3', datetime.now()+timedelta(minutes=1), "Take control of IoT device"],
        ['3', datetime.now()+timedelta(minutes=2), "Conduce DDoS on multiple
        targets"]]

df = pd.DataFrame(data, columns=['case:concept:name', 'time:timestamp', '
    concept:name'])
```

Figure 4.12: Creation of a Pandas Dataframe for the generation of a dummy event log.

	case:concept:name	time:timestamp	concept:name
0	0	2022-12-14 17:50:58.712382	4624 - An account was successfully logged on
1	0	2022-12-14 17:51:58.712382	4625 - An account failed to log on
2	0	2022-12-14 17:52:58.712382	4626 - User/Device claims information
3	0	2022-12-14 17:53:58.712382	4627 - Group membership information.
4	0	2022-12-14 17:54:58.712382	4634 - An account was logged off
5	1	2022-12-14 17:50:58.712382	Eavesdropping
6	1	2022-12-14 17:50:58.712382	Malware
7	1	2022-12-14 17:50:58.712382	Exploit IoT vulnerabilities
8	1	2022-12-14 17:51:58.712382	Take control of IoT device
9	1	2022-12-14 17:52:58.712382	Conduce DDoS on multiple targets
10	2	2022-12-14 17:50:58.712382	Malware
11	2	2022-12-14 17:50:58.712382	Eavesdropping
12	2	2022-12-14 17:50:58.712382	Exploit IoT vulnerabilities
13	2	2022-12-14 17:51:58.712382	Take control of IoT device
14	2	2022-12-14 17:52:58.712382	Conduce DDoS on multiple targets
15	3	2022-12-14 17:50:58.712382	Exploit IoT vulnerabilities
16	3	2022-12-14 17:50:58.712382	Malware
17	3	2022-12-14 17:50:58.712382	Eavesdropping
18	3	2022-12-14 17:51:58.712382	Take control of IoT device
19	3	2022-12-14 17:52:58.712382	Conduce DDoS on multiple targets

Figure 4.13: Example of generated Pandas Dataframe containing log instances.

As stated before in the design section, the next step is the conversion of the Pandas Dataframe into an event log object. Our goal is to convert

the DataFrame into a XES file. To do this, we can use PM4PY [17], which is a Process Mining package for Python. It implements the latest, most useful, and extensively tested methods of Process Mining. The code in Figure 4.14 shows how to convert the obtained Pandas Dataframe into the PM4PY internal event data object type. To do this, we refer to the `format_dataframe` function available in PM4PY. By default, the converter transform the Dataframe into an Event Log object. In this example, we observe three columns: `case:concept:name` for the ID, `concept:name` for the name of the current activity, and `time:timestamp` for the timestamp. The event log is then exported into an external XES file, so that it can be used as input data for the Process Mining algorithm. Exporting an Event Log object to a XES file is fairly straightforward with PM4PY. The last line of code in Figure 4.14 depicts this functionality. The `event_log` is assumed to be an Event Log object. The exporter also accepts an Event Stream or DataFrame object as input. As a final step, the log can be exported as XES file using the `write_xes` PM4PY function.

```
event_log = pm4py.format_dataframe(df, case_id='case:concept:name',
    activity_key='concept:name', timestamp_key='time:timestamp')

# export log in .xes file
pm4py.write_xes(event_log, 'xesTest.xes')
```

Figure 4.14: Conversion of a Dataframe in an Event Log and exportation in XES format.

In addition, PM4PY uses the Pandas library to enable the conversion of an event log into a CSV file. Hence, a XES event log may be first transformed into a Pandas DataFrame with the function `convert_to_dataframe`, and then the translation into CSV can be done with the `to_csv` function from Pandas. Also the vice-versa is possible: CSV event logs can be easily read through Pandas functions, and then a XES log can be obtained using `convert_to_event_log` from PM4PY.

Now we can apply the Process Mining algorithm on the dummy logs that we just generated. We run the experiments to test our methodology using PM4PY. In this library an implementation of the Inductive Miner is

offered. Two process models can be derived using the IM: Petri Net and Process Tree. To mine a Process Tree, as shown below in Figure 4.15, a log is read, the Inductive Miner is applied and the PT along with the initial and the final marking are found. The same reasoning may be applied to find a Petri Net as an alternative. In this case the 'dummy' log `xesTest.xes` is taken as input. Firstly, the log is read and then the IM algorithm is applied. As we said, besides the Process Tree, the provided code can also be used to obtain a Petri Net. The last two lines of code are responsible for the PT visualization. The resulting tree is shown in Figure 4.16. If necessary, it is also possible to convert a Process Tree into a Petri Net.

```
# A log is read, IM is applied and the Petri net with the initial and the
    final marking are found
log = pm4py.read_xes(os.path.join("xesTest.xes"))
net, initial_marking, final_marking = pm4py.discover_petri_net_inductive(log)

# Visualization PT
tree = pm4py.discover_process_tree_inductive(log)
pm4py.view_process_tree(tree)
```

Figure 4.15: Use of Inductive Miner on the dummy event logs to generate a Process Tree.

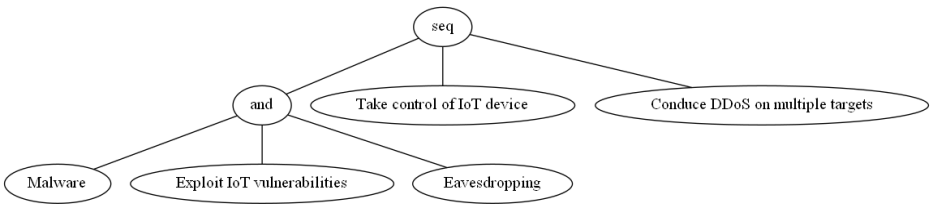

Figure 4.16: Process Tree generated by using the implementation of the Inductive Miner on dummy event logs.

Now it is possible to follow the procedure presented before, to transform the obtained Process Tree into an Attack Tree. In the previous section we outlined the main characteristics and translation rules about the PT that is mapped into the final AT. As a result from the application of the translation rules, we get an Attack Tree generated from the 'dummy' logs. Figure 4.17 depicts the result of the translation process. Note that the sequence operator is used to transform the relationship into a parent-child

one. Finally, the AND operator becomes the conjunction one, while the root is now the silent activity τ.

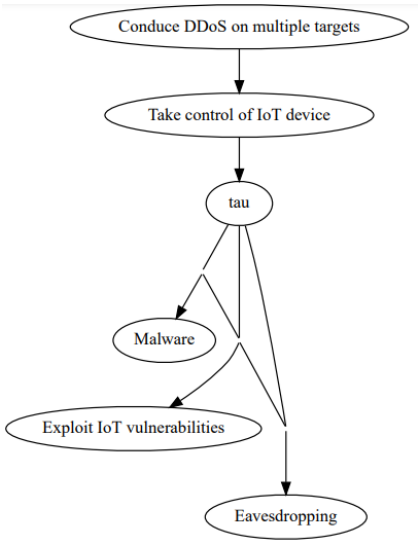

Figure 4.17: Resulting AT starting from generated dummy logs and passing through the PT shown in section 4.2

The tool's last component is a translation script designed to convert the obtained XML Attack Tree into the same one but in `bbt` format, which is suitable for the RisQFLan security framework described in chapter 3. This is accomplished by using the rules of semantics that were just discussed. The attack nodes are defined, together with the relationships between them. What we obtain is a `txt` file containing the RisQFLan code. As a final step, the conversion into a `bbt` file can be done. The code depicted in Figure 4.18 can be passed to RisQFLan as input, so that the Attack Tree can be viewed in this framework as well.

```
begin model AttackTree

  begin attack nodes
    tau
    ConduceDDoSonmultipletargets
    TakecontrolofIoTdevice
    ExploitIoTvulnerabilities
    Eavesdropping
  end attack nodes

  begin attack diagram
    ConduceDDoSonmultipletargets -> {TakecontrolofIoTdevice}
    TakecontrolofIoTdevice -> {tau}
    tau -> {ExploitIoTvulnerabilities}
    tau -> {Eavesdropping}
  end attack diagram

end model
```

Figure 4.18: Code referring to the XML attack tree in section 3.1 and suitable for the RisQFLan tool.

After giving an overview of the tool proposed in this thesis, in the next chapter we will provide some actual case studies. The last example uses real logs, as opposed to the prior three that use the external tool OFMC, which allows one to obtain event logs starting from attack traces.

Chapter 5

Case studies

In the prior chapter, an introduction to the design was necessary before the case studies were given. The Opensource Fixedpoint Model Checker (OFMC) tool was also introduced in order to obtain an event log starting with an attack trace generated by this tool. OFMC is used in our first three examples. On the other hand, the last case study starts directly from a real system's event log and it also serves as a 'bad example'. Indeed, the selected log prevents one from constructing the system's final Attack Tree from the discovered Process Tree. The reasons will be addressed at the end of this chapter.

5.1 First example: attack trace concerning a single attack

5.1.1 Attack trace generation

To be clearer, now we make an example regarding the functioning of the tool. For the sake of simplicity, in this first case study we generated an OFMC attack trace that takes into account only one attack to the protocol. However, later we will analyze cases involving attack traces with multiple attacks, so that the final Attack Tree can be as complete as possible. To begin with, we introduce a simple key-exchange protocol that establishes a

shared symmetric key between two parties, i.e. Alice and Bob, that do not have a security relationship so far. We want to protect the transmission of a secret key KAB shared between A and B. The problem in this example is that between the transmission there is an intruder called I. With this aim, we assume that all the agents (including the intruder) initially have a shared key with the server S. We may think that S provides wireless access, but everyone who wants to use it has to first register, for example with an offline operation that involes the installation of username and secret shared key between the agent and the server. Therefore, we assume that S has a strong shared key sk(A, S) with every user A, i.e., chosen by a cryptographic random-number generator. Figure 5.1 shows the example using the KeyEx protocol. Note that AnB language uses the notation $\{|Mes|\}K$ for the symmetric encryption of a given message Mes with the key K.

```
Protocol: KeyEx

Types: Agent A,B,s;
       Symmetric_key KAB;
       Function sk

Knowledge:
  A: A,B,s,sk(A,s);
  B: A,B,s,sk(B,s);
  s: A,B,s,sk(A,s),sk(B,s)

Actions:

A->s: A,B
s->A: {| KAB |}sk(A,s), {| KAB |}sk(B,s)
A->B: A,{| KAB |}sk(B,s)
```

Figure 5.1: Example in AnB language about key exchange protocol.

Let's start introducing the `types` section. Here A, B and KAB, together with other identifiers that starts with upper case letters, are variables. They are placeholders for a concrete value, e.g. the name of the agent. Identifiers that start with lower case letters, like s, are called constants. Variables and constants that are declared to be of type `Agent` are called roles. It is important to use variables and constants: variables of type Agent cannot be instantiated arbitrarily with agent names. This includes the special agent i, that is the intruder.

Moreover, for each role of the protocol, its initial knowledge must be specified. This is done in the `knowledge` section. Note that the elements in the knowledge part must contain only variables of type agent.
Finally, let's analyze the `action` section: here the parties' message exchanges are described. First of all, A tells to the server S that he would like to communicate with B. Then, S creates a new symmetric key and it also generates two encryptions: the first using the shared key with A, and the other one exploiting the shared key with B. The two messages are then encrypted and sent to A, which can only decrypt the first of the two. A is supposed to forward this second message to B, who has the necessary knowledge to decrypt it. Therefore, all agents have sufficient knowledge to produce all necessary messages and end up with a copy of the shared key KAB at least in a run, when the intruder does not interfere. Note that A cannot decrypt the second part of the message received from S.
Usually, cryptographers associate with 'symmetric encryption' only the pure encryption, without any protection of integrity such like the MAC (Message Authentication Code). This pure encryption sometimes can be vulnerable to the intruder that manipulates the ciphertext and changes the encrypted text. By doing so, the recipient cannot detect that a change was done. Therefore, AnB models $\{|Mes|\}K$ as a primitive which includes integrity. When A receives the two encrypted messages from S, he will decrypt the first one. Indeed, the integrity mechanism of the primitive allows to check that the received message is encrypted correctly with the right symmetric key $sk(A, s)$. If I sends other messages that are not of the form $\{|Mes|\}sk(A, s)$, A will detect a problem and will refuse them.

If we run this OFMC example, we otbain the attack trace depicted in Figure 5.2. The attack in question is a violation against weak authentication. In the attack trace, looking at the comments that OFMC provides as part of the Reached State comment, we can note three facts. They are often helpful in understanding an authentication attack, because they reflect what the honest agents 'think'. Here, the intruder chose two random agent names x29,x401 and sent them to S, who created a new shared key KAB(1) for these two. Then, I sent this message to x401, but claiming to be

x27, namely a different person than who the server generated the key for. To x401 this message looks like a correct step 3 of the protocol, so it will believe that KAB(1) is a shared key with x27. The intruder did not discover any secrets in this case. However, he violated the authentication: S and recipient x401 disagree on who is playing role A in this session, namely with who the key is shared with. Therefore, this is a violation of the goal `B authenticates s on A,KAB`.
To be clearer, we say that we have a violation of weak authentication if there is a request fact without a matching witness fact. If we consider a goal with the form `B authenticates A on M`, the witness fact reflects the point of view of A. On the other hand, the request fact reflects the point of view of B. Thus, this goal should be used if the protocol ensures the authentic communication of a message Mes from A to B. Therefore, if B finishes believing that A has sent message Mes for him, this is indeed an attack. This also includes a case in which A has meant the message for somebody else, as the one we just analyzed.

```
  ATTACK_FOUND
GOAL
  weak_auth
...
ATTACK TRACE
i -> (s,1): x29,x401
(s,1) -> i: {|KAB(1)|}_(sk(x29,s)),{|KAB(1)|}_(sk(x401,s))
i -> (x401,1): x27,{|KAB(1)|}_(sk(x401,s))


% Reached State:
%
% request(x401,s,pBsKABA,KAB(1),x27,1)
% witness(s,x29,pAsKABB,KAB(1),x401)
% witness(s,x401,pBsKABA,KAB(1),x29)
% ...
```

Figure 5.2: Attack trace generated from the key exchange protocol example.

We have now generated an attack trace regarding a specific protocol. However, for the purpose of this thesis we need some event logs to perform Process Mining and obtain the Attack Tree of the system. To do this, we implemented some slight modifications to the AnB example previously introduced in Figure 5.1. The aim is to produce some significant labels for the conversion of the attack trace into an event log. To achieve this, a new `Type` offered by OFMC, called `Number`, was used. This type allows one to define some 'labels' that can be added in the list of `Knowledge` of the

agents. Keep in mind that for the labeling process to be effective, each agent participating must know all of the labels. The updated code about the Key-exchange protocol is shown in Figure 5.3, while the resulting labeled attack trace can be seen in Figure 5.4.

```
Protocol: KeyEx
# 2nd version: Adding labeling for the purpose of event log generation.

Types:
  Agent A,B,s;
  Symmetric_key KAB;
  Function sk;
  Number aSendsTheParticipants, serverSendsEncryptedKeyToA, aSendsEncryptedPartOfKeyToB

Knowledge:
  A: A,B,s,sk(A,s), aSendsTheParticipants, serverSendsEncryptedKeyToA, aSendsEncryptedPartOfKeyToB;
  B: A,B,s,sk(B,s), aSendsTheParticipants, serverSendsEncryptedKeyToA, aSendsEncryptedPartOfKeyToB;
  s: A,B,s,sk(A,s),sk(B,s), aSendsTheParticipants, serverSendsEncryptedKeyToA, aSendsEncryptedPartOfKeyToB

Actions:

A->s: aSendsTheParticipants, A,B
# s creates key KAB
s->A: serverSendsEncryptedKeyToA, {| KAB |}sk(A,s), {| KAB |}sk(B,s)
A->B: aSendsEncryptedPartOfKeyToB, A,{| KAB |}sk(B,s)
```

Figure 5.3: Example in AnB language about key exchange protocol with labeling.

```
ATTACK TRACE:
i -> (s,1): A Sends The Participants To S,x29,x401
(s,1) -> i: Server Sends Encrypted Key To A,{|KAB(1)|}_(sk(x29,s)),{|KAB(1)|}_(sk(x401,s))
i -> (x401,1): A Sends Encrypted Part Of Key To B,x27,{|KAB(1)|}_(sk(x401,s))

#% Reached State:
#%
#% request(x401,s,pBsKABA,KAB(1),x27,1)
#% witness(s,x29,pAsKABB,KAB(1),x401)+
#% witness(s,x401,pBsKABA,KAB(1),x29)
#% ...
```

Figure 5.4: Labeled attack trace generated from the key exchange protocol example.

We point out that, even though we reached the same attack trace as before, now there are labels that describe what is happening in each line, which can be helpful for the creation of a labeled event log.

5.1.2 Translation from attack trace to event log

Proceeding with the work, given as input the obtained attack trace, we can convert it and generate a Dataframe of logs using the Pandas library from Python. Afterwards, it is possible to export the Dataframe as XES log file for the Process Mining. The specific columns of data required in

the log are the same as those in the subsection 5.4.1 about the 'dummy example' that we introduced previously. To be precise, we need:

- IDs: an identification number of a single activity in the logs, to distinguish it from other events.

- Timestamps of activities: all events have a date and time associated. This is useful when analyzing performance related properties, e.g., the waiting time between two activities. If an event log contains timestamps, then the ordering in a trace should respect these timestamps, i.e. they should be nondescending in the event log.

- Activity description: this field includes the label that describes what this line of log is about, together with the content of the message exchanged between the two parties. The content of the message is useful to understand why the weak authentication attack is happening and what the intruder is doing. In this specific case, the intruder is faking his real identity: he claims to be x29 with the server, but to the receiver x401 he declarates to be x27. Here the authentication is broken.

- Information/content: this column contains the label to describe the single atomic activity.

To isolate all the information, a translation script was written so that the Pandas Dataframe could be generated. The script can be seen in Figure 5.5. This is suitable to convert any kind of attack trace generated by the OFMC tool, even the most complex ones.

```
# convert the initial attack trace generated by OFMC into csv
my_file = 'Attack traces/AttackTraceExample2.log'
base = os.path.splitext(my_file)[0]
os.rename(my_file, base + '.csv')

# convert the OFMC attack trace into Dataframe
df = pd.read_csv('Attack traces/AttackTraceExample2.csv')
print(df)

listaDF = df.values.tolist()
print(listaDF)

# loop every line of the trace to obtain single informations
for data in listaDF:
    res = str(data).split("'", 1)[1]
    res1 = res.split(":")[1]
    label = res1.split(",")[0]
    label = label.split("'")[0] # atomic activity label
    res1 = str(data).split("'", 1)[1]
    senderA = res1.split("-")[0] # sender
    res2 = str(data).split(">", 1)[1]
    receiverB = res2.split(":")[0] # receiver
    res3 = str(data).split(":")[1]
    # CASE 1: sender + receiver + message content
    content = senderA + " -> " + receiverB + ", " + res3.split("'")[0]
    # content = senderA + " -> " + receiverB # CASE 2: sender + receiver
    # content = senderA + ", " + res3.split("'")[0] # CASE 3: sender + content
    time = datetime.now()+timedelta(minutes=i)
    updatedRows.append([str(time), content, "0", label])
    i = i+1

# creation of the Dataframe with the right data
updatedPD = pd.DataFrame(updatedRows, columns = ['time:timestamp','concept:
    activity','@@index','case:concept:info'])

print("\nNew dataframe with proper columns: ")
print(updatedPD)
```

Figure 5.5: Translation script: from attack trace to event log.

To be precise, a conversion from `log` to `csv` is needed. Afterwards, the trace can be read by the Pandas library as a Dataframe, and subsequently converted into a List of List using the `tolist()` function. We can now loop the list and isolate only the strings of data that we need, to create a proper event log containing the following information: `timestamp`, `activity`, `index`, `info`. In this case, the 'info' is the description of the atomic activity, i.e. the content of the exchanged message. On the other hand, the "activity" column can contain different elements, because we chose to split and analyze three distinct cases. Indeed, the 'activity' field can contain:

- **Case 1**: the sender, the receiver and the content of the message;

- **Case 2**: the sender and the receiver of the message;
- **Case 3**: the sender and the content of the message.

The three different scenarios are outlined in lines 24, 26 and 27 of Figure 5.5. Then, we can put all the final information together in a Dataframe, e.g. `updatedPD`, that will become the final event log when we export it to XES.
In Figure 5.6 the single components of the attack trace are summarized, while Figure 5.7 shows the obtained Dataframe containing the proper columns of data generated from the key-exchange attack trace.

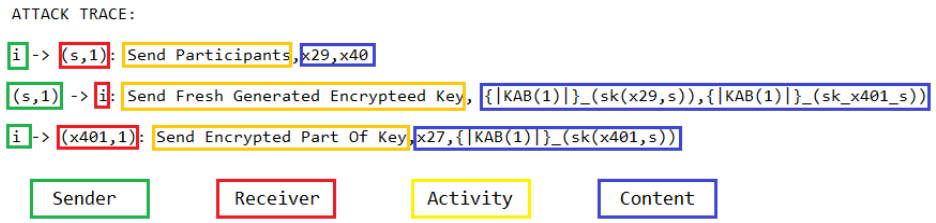

Figure 5.6: Translation summary from attack trace to event log.

	time:timestamp	concept:activity	@@index	case:concept:info
0	2022-12-16 11:39:14.110491	Send Participants,x29,x401, i -> (s,1)	0	Send Participants
1	2022-12-16 11:40:14.110491	Send Fresh Generated Encrypted Key,{|KAB(1)|}...	0	Send Fresh Generated Encrypted Key
2	2022-12-16 11:41:14.110491	Send Encrypted Part Of Key,x27,{|KAB(1)|}_(sk...	0	Send Encrypted Part Of Key

Figure 5.7: Resulting Dataframe of logs starting from the key exchange attack trace.

Exporting a XES event log starting from the generated Pandas Dataframe is the last step of this phase. As we can see in Figure 5.8, it is possible to carry out the conversion by employing the PM4PY library.

```
event_log = pm4py.format_dataframe(dataframe, case_id='case:concept:name',
    activity_key='concept:name', timestamp_key='time:timestamp')
pm4py.write_xes(event_log, 'createdXes.xes')   # exports a xes log
```

Figure 5.8: Conversion from Pandas Dataframe to XES event log.

5.1.3 From event log to Process Tree and Attack Tree

The next step is Process Mining: our aim is to obtain a Process Tree starting from the event logs. To do this, we can follow the rules outlined in Section 2.2. The code that depicts this step can be seen in Figure 5.9.

```
# The inductive miner is applied to the log and the Petri net along with the
    initial and the final marking are found
log = pm4py.read_xes(os.path.join("createdXes.xes"))
net, initial_marking, final_marking = pm4py.discover_petri_net_inductive(log)

# Visualization of PT
tree = pm4py.discover_process_tree_inductive(log)
pm4py.view_process_tree(tree)
```

Figure 5.9: Read of the log and generation of the Process Tree.

Basically, we can employ the PM4PY functions `read_xes` and `discover_process_tree_inductive` to respectively read the analyzed log as input and discover the related Process Tree. Afterwards it is also possible to visualize it through `view_process_tree`. In addition to Process Trees, another type of graph that is possible to discover is the Petri Net. This is possible with the `discover_petri_net_inductive` function.

The resulting Process Tree is showed in Figure 5.10. The depicted tree refers to Case 1 in which we take into account sender, receiver and content of the message. To make the tree more readable, hereinafter we explain the meaning of each node:

- To begin the message exchange, A sends the participants to the server S. The content of the message is `x29, x401`, i.e., the first term is claimed to be A, while the second is B.
- The server now generates a key `KAB` for the message exchange and sends it to A. The content of the message is $\{|\texttt{KAB(1)}|\}_{\texttt{(sk(x29,s))}}$, $\{|\texttt{KAB(1)}|\}_{\texttt{(sk(x401,s))}}$, namely S creates two encryptions: one using the shared key with A, and another one using the shared key with B.
- A sends to B his encrypted part of the key, but claiming to be x27 and not x29 as he told to the server. We can spot this problem looking at

the content of the exchanged message: `x27, {|KAB(1)|}_(sk(x401,s)).` Indeed, the first term in the message describes who A is, and in this case we can read x27 and not x29. The attack takes place in this precise step: the intruder did not find out any secret, but he managed to break the authentication.

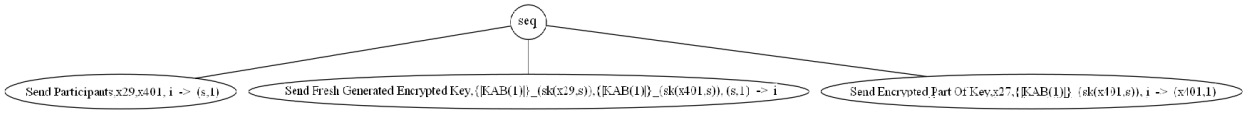

Figure 5.10: Process Tree Tree about single authentication attack starting from OFMC attack traces (Case 1).

Instead, Figure 5.11 depicts the Process Tree that refers to Case 2, i.e. the activity nodes contain only sender and receiver. Finally, Figure 5.12 shows Case 3 where the Process Tree's nodes take into account only sender and content of the message.

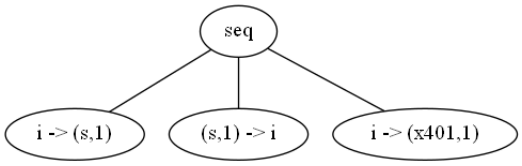

Figure 5.11: Process Tree Tree about single authentication attack starting from OFMC attack traces (Case 2).

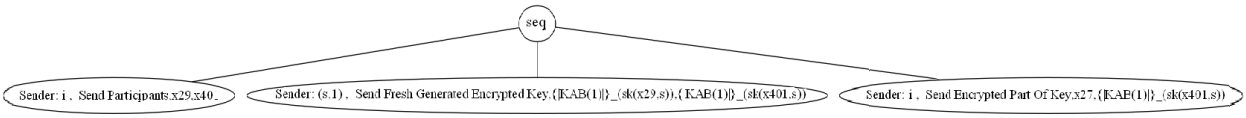

Figure 5.12: Process Tree Tree about single authentication attack starting from OFMC attack traces (Case 3).

The Process Tree nodes can now be translated into Attack Tree nodes, to generate the final Attack Tree of the system. To do this, we follow the translation rules introduced in Section 4.2. As we said before, to be able to see different granularity levels, we decided to consider three different cases for the 'activity' field in the event log. Because of this, we generated

three distinct Attack Trees, each concerning the Process Trees that we just showed. The one containing sender, receiver and content of the message can be seen in Figure 5.13 (Case 1), while the one only about sender and receiver is shown in Figure 5.14 (Case 2). Finally, the AT containing only sender and content of the message is depicted in Figure 5.15 (Case 3). The trees are all about the same event log, but they allow us to see different data from multiple perspectives.

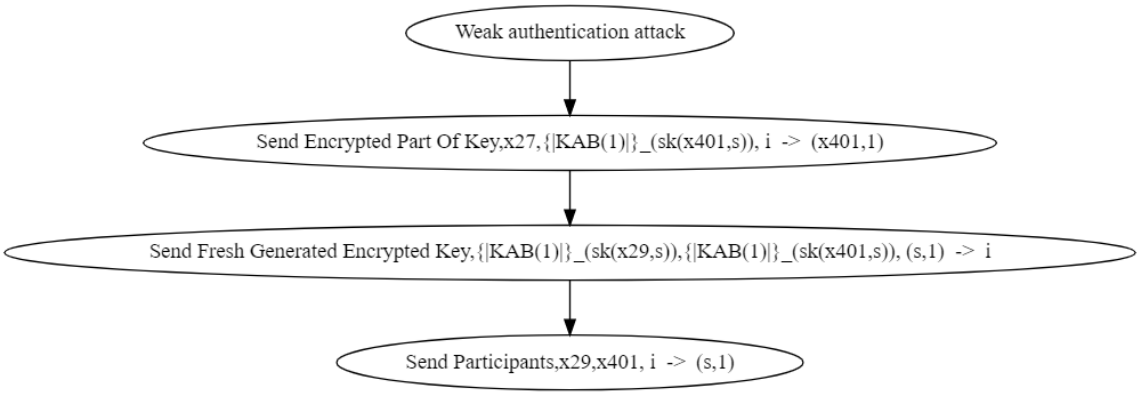

Figure 5.13: Resulting Attack Tree about single authentication attack containing sender, receiver and message content (Case 1).

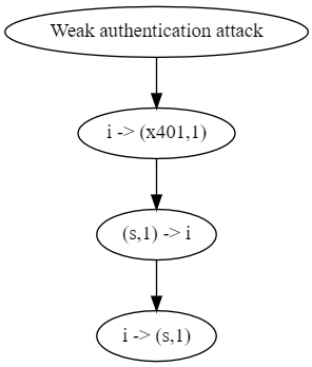

Figure 5.14: Resulting Attack Tree about single authentication attack containing sender and receiver (Case 2).

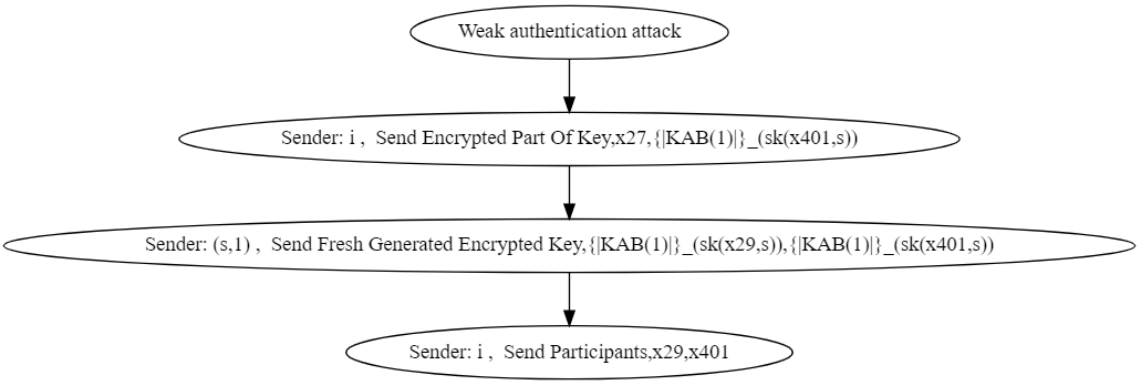

Figure 5.15: Resulting Attack Tree about single authentication attack containing sender and message content (Case 3).

5.1.4 Generation of RisQFLan's code

Lastly, to translate the three obtained Attack Trees from XML into `bbt`, a final translation script was written. This extension is supported by the RisQFLan tool, previously introduced in Section 3.2. First of all, the attack model was created and subsequently the attack nodes were detected. To identify them, we iterated on all the nodes involved in the XML tree, both parents and children. Finally, the types of relationship between them were specified so that the code could be generated and could be given as input to the security framework, to generate the tree for RisQFLan. To be able to obtain the relationships, another iteration was done. If the `parentRelationship` of a given node is `disjunction`, it is translated into an 'OR' relationship, otherwhise if we have a `conjunction`, we obtain an 'AND'. The obtained code of the Attack Tree about Case 1, i.e. the one that includes content, sender and receiver, can be seen below in Figure 5.16. The generation of RisQFLan code for the other two cases follows the same steps and reasoning.

```
begin model AttackTree

  begin attack nodes
    tau
    Weakauthenticationattack
    Sender:i,SendEncryptedPartOfKey,x27,{|KAB(1)|}_(sk(x401,s))
    Sender:(s,1),SendFreshGeneratedEncryptedKey,{|KAB(1)|}_(sk(x29,s)),{|KAB
        (1)|}_(sk(x401,s))
    Sender:i,SendParticipants,x29,x401
  end attack nodes

  begin attack diagram
    tau -> {Weakauthenticationattack}
    Weakauthenticationattack -> {Sender:i,SendEncryptedPartOfKey,x27,{|KAB(1)
        |}_(sk(x401,s))}
    Sender:i,SendEncryptedPartOfKey,x27,{|KAB(1)|}_(sk(x401,s)) -> {Sender:(s
        ,1),SendFreshGeneratedEncryptedKey,{|KAB(1)|}_(sk(x29,s)),{|KAB(1)|}_(
        sk(x401,s))}
    Sender:(s,1),SendFreshGeneratedEncryptedKey,{|KAB(1)|}_(sk(x29,s)),{|KAB
        (1)|}_(sk(x401,s)) -> {Sender:i,SendParticipants,x29,x401}
  end attack diagram

end model
```

Figure 5.16: RisQFLan code for the Attack Tree containing sender, receiver and content of the message.

We emphasize the resulting trees' simplicity as a way to wrap up this initial case study. This is the result of a trace that considers one single attack at the authentication protocol. However, the same logic and reasoning can also be applied for case studies that are trickier. For this reason, hereinafter we include other examples of attack traces involving multiple attacks to the same OFMC input protocol.

5.2 Second example: attack trace concerning multiple attacks

For the sake of simplicity, we used the previous case study to generate a trivial Attack Tree with sequential shape. To make this tool more realistic, we introduce now an example in which another attack trace generated from OFMC is transformed into an event log and then the previous procedure is followed. The distinction in this case study is that, unlike the prior one, this trace considers various attacks against the same protocol, and not only one. For this example we produced an OFMC trace concern-

ing the Needham–Schroeder Protocol (NSPK Protocol). The AnB code that allowed the generation of the trace is depicted in Figure 5.17, while the obtained trace can be seen in Figure 5.18. The Needham–Schroeder Public-Key Protocol is based on public-key cryptography. It aims to establish a session key between two parties on a network, typically to protect further communication, but in its proposed form is considered insecure.

```
Protocol: NSPK

Types: Agent A,B;
       Number NA,NB;
       Function pk

Knowledge: A: A,pk,inv(pk(A)),B;
           B: B,pk,inv(pk(B))

Actions:
# A creates NA
A->B: {NA,A}(pk(B))
# B creates NB
B->A: {NA,NB}(pk(A))
A->B: {NB}(pk(B))

Goals:

B authenticates A on NA
A authenticates B on NB
NA secret between A,B
NB secret between A,B
```

Figure 5.17: Code for OFMC for the generation of the attack trace about protocol NSPK.

As we can see in Figure 5.17, in this case we take into account two agents, i.e. A and B, and an encryption function called pk. Indeed, the communication happens only between A and B without the presence of a third party. The key $pk(A)$ is symmetric and known only by A, while $pk(B)$ only by B. Moreover, in this case study there are two new elements called nonces: they are a way to ensure recentness in protocols of challenge-response. To be precise, nonces are random numbers chosen by one or more parties. They abbreviate 'number once', i.e. they should be used only once. The point is that, if another party has to include the nonce in a response, the creator of the nonce can be sure that the response is no older than the nonce it contains. The value of these guarantees depends on the cryptographic operations in which the nonce is used.
First of all, in the code depicted in Figure 5.17, A begins creating his nonce NA and sends it together with A, encrypting it with the encryption func-

tion of B, i.e. $pk(B)$. Subsequently, B does the same creating his nonce NB and sending it together with NA, encrypting everything with $pk(A)$. Finally, A can decrypt what he just received and can send NB encrypted with $pk(B)$. Basically, A re-encrypts it and sends it back to verify if that he is still alive and that he holds the key.

```
ATTACK TRACE:
**ALLIN:
########### Attack 1 ########## (step1, step1, step1, step2, step2, step3)
weak_auth*(x20,1) -> i: Sending encrypted fresh nonce NA,{NA(1),x20}_(pk(x26))
(x702,2) -> i: Sending encrypted fresh nonce NA,{NA(2),x702}_(pk(i))
i -> (x701,2): Sending encrypted fresh nonce NA,{NA(2),x702}_(pk(x701))
(x701,2) -> i: Sending encrypted nonce NA + new fresh NB,{NA(2),NB(3)}_(pk(x702))
i -> (x702,2): Sending encrypted nonce NA + new fresh NB,{NA(2),NB(3)}_(pk(x702))
(x702,2) -> i: Sending encrypted nonce NB,{NB(3)}_(pk(i))
i -> (x701,2): Sending encrypted nonce NB,{NB(3)}_(pk(x701))

########### Attack 2 ########## (step1, step1, step2, step1, step1, step2, step2, step3, step3)
weak_auth*(x304,1) -> i: Sending encrypted fresh nonce NA,{NA(1),x304}_(pk(x28))
i -> (x28,1): Sending encrypted fresh nonce NA,{NA(1),x304}_(pk(x28))
(x28,1) -> i: Sending encrypted nonce NA + new fresh NB,{NA(1),NB(2)}_(pk(x304))
(x802,2) -> i: Sending encrypted fresh nonce NA,{NA(3),x802}_(pk(i))
i -> (x801,2): Sending encrypted fresh nonce NA,{NA(3),x802}_(pk(x801))
(x801,2) -> i: Sending encrypted nonce NA + new fresh NB,{NA(3),NB(4)}_(pk(x802))
i -> (x802,2): Sending encrypted nonce NA + new fresh NB,{NA(3),NB(4)}_(pk(x802))
(x802,2) -> i: Sending encrypted nonce NB,{NB(4)}_(pk(i))
i -> (x801,2): Sending encrypted nonce NB,{NB(4)}_(pk(x801))

########### Attack 3 ########## (step1, step1, step2, step2, step3, step1, step1, step2, step2, step3, step3)
weak_auth*(x304,1) -> i: Sending encrypted fresh nonce NA,{NA(1),x304}_(pk(x28))
i -> (x28,1): Sending encrypted fresh nonce NA,{NA(1),x304}_(pk(x28))
(x28,1) -> i: Sending encrypted nonce NA + new fresh NB,{NA(1),NB(2)}_(pk(x304))
i -> (x304,1): Sending encrypted nonce NA + new fresh NB,{NA(1),NB(2)}_(pk(x304))
(x304,1) -> i: Sending encrypted nonce NB,{NB(2)}_(pk(x28))
(x902,2) -> i: Sending encrypted fresh nonce NA,{NA(4),x902}_(pk(i))
i -> (x901,2): Sending encrypted fresh nonce NA,{NA(4),x902}_(pk(x901))
```

Figure 5.18: Attack trace containing multiple attacks to NSPK protocol.

As confirmed by the obtained attack trace, the protocol is vulnerable to a replay attack. If an attacker uses an older, compromised key, he could replay his message to B who will accept it, being unable to tell that the key is not fresh. A part of the obtained attack trace can be seen in Figure 5.18.

5.2.1 Translation from attack trace to event log

Similarly to the former example, the attack trace was first converted into a Pandas Dataframe using the same translation script that was depicted in Figure 5.5. Figure 5.19 shows the obtained Dataframe. For simplicity, in this case study we took into account the first 4 attacks that appear in the trace, but it is possible to consider any preferred number. We chose to limit the selected attacks from the trace because, for the sake of

our explanation, a larger number would result in a wider and more chaotic Attack Tree.

	time:timestamp	concept:activity	@@index	case:concept:info
0	2022-12-19 16:34:25.061698	Sending encrypted fresh nonce NA,{NA(1),x20}_...	0	Sending encrypted fresh nonce NA
1	2022-12-19 16:36:25.062681	Sending encrypted fresh nonce NA,{NA(2),x702}...	0	Sending encrypted fresh nonce NA
2	2022-12-19 16:38:25.062681	Sending encrypted fresh nonce NA,{NA(2),x702}...	0	Sending encrypted fresh nonce NA
3	2022-12-19 16:40:25.062681	Sending encrypted nonce NA + new fresh NB,{NA...	0	Sending encrypted nonce NA + new fresh NB
4	2022-12-19 16:42:25.062681	Sending encrypted nonce NA + new fresh NB,{NA...	0	Sending encrypted nonce NA + new fresh NB
5	2022-12-19 16:44:25.062681	Sending encrypted nonce NB,{NB(3)}_(pk(i))\n,...	0	Sending encrypted nonce NB
6	2022-12-19 16:46:25.062681	Sending encrypted nonce NB,{NB(3)}_(pk(x701))...	0	Sending encrypted nonce NB
7	2022-12-19 16:49:25.062681	Attack against nspk	0	
8	2022-12-19 16:50:25.063680	Sending encrypted fresh nonce NA,{NA(1),x304}...	1	Sending encrypted fresh nonce NA
9	2022-12-19 16:51:25.063680	Sending encrypted fresh nonce NA,{NA(1),x304}...	1	Sending encrypted fresh nonce NA
10	2022-12-19 16:52:25.063680	Sending encrypted nonce NA + new fresh NB,{NA...	1	Sending encrypted nonce NA + new fresh NB
11	2022-12-19 16:53:25.063680	Sending encrypted fresh nonce NA,{NA(3),x802}...	1	Sending encrypted fresh nonce NA
12	2022-12-19 16:54:25.063680	Sending encrypted fresh nonce NA,{NA(3),x802}...	1	Sending encrypted fresh nonce NA
13	2022-12-19 16:55:25.063680	Sending encrypted nonce NA + new fresh NB,{NA...	1	Sending encrypted nonce NA + new fresh NB
14	2022-12-19 16:56:25.063680	Sending encrypted nonce NA + new fresh NB,{NA...	1	Sending encrypted nonce NA + new fresh NB
15	2022-12-19 16:57:25.063680	Sending encrypted nonce NB,{NB(4)}_(pk(i))\n,...	1	Sending encrypted nonce NB
16	2022-12-19 16:58:25.063680	Sending encrypted nonce NB,{NB(4)}_(pk(x801))...	1	Sending encrypted nonce NB
17	2022-12-19 17:00:25.063680	Attack against nspk	1	
18	2022-12-19 17:01:25.064679	Sending encrypted fresh nonce NA,{NA(1),x304}...	2	Sending encrypted fresh nonce NA
19	2022-12-19 17:02:25.064679	Sending encrypted fresh nonce NA,{NA(1),x304}...	2	Sending encrypted fresh nonce NA

Figure 5.19: Dataframe containing log entries concerning attacks agains NSPK.

Afterwards, the Dataframe was exported into a XES event log using the PM4PY Python library, as previously shown in Figure 5.8. Also in this case study, the relevant data from the log concern the following information: ID, timestamp, activity label and information. Now, the obtained XES file can be given in input to the next step, i.e. the Process Mining algorithm.

5.2.2 From event log to Process Tree and Attack Tree

Continuing our process, we can use the functions `discover_process_tree_inductive` and `view_process_tree` from PM4PY, like we did before in Figure 5.9. By performing this and using the event log as input, we can discover and visualize the related Process Tree of the system. In addition, we differentiated three different scenarios for the 'activity' field displayed in the tree's nodes: first, the activity could include include the sender, the recipient and the message's content; second, it could include only the sender and receiver; and third, it could only include the message's sender and

content. As a result, we were able to create three alternative Process Trees. Figures 5.20 and 5.21 show the first two Process Trees, as we decided to omit the third one for space reasons.

Figure 5.20: Process Trees about case 1 (activity nodes containing content of the message, sender and receiver).

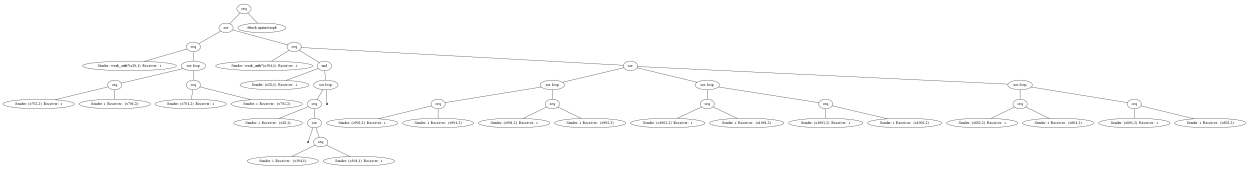

Figure 5.21: Process Trees about case 2 (activity nodes containing sender and receiver).

The Process Tree nodes can now be transformed into Attack Tree nodes to construct the system's final Attack Tree. To do this, we followed the translation rules previously outlined in Section 4.2. We attempted to develop three related Attack Trees, one for each scenario, based on the three Process Trees we just introduced. Figure 5.22 displays the scenario about message's sender, receiver, and content (case 1), while Figure 5.23 depicts the one only about message's sender and content (Case 3). Even though the trees are all about the same event log, they allow us to view different pieces of information from different perspectives. Regarding the second scenario, be aware that the Attack Tree is missing. This happened because the second Process Tree contained several 'xor loops' that could not be changed into other appropriate operators for the Attack Tree. As a result, the transformation in this second case was not feasible. The final chapter of this thesis, which discusses the results and potential future improvements, tries to address issue. We will highlight that one alternative approach may be to consider the adoption of a different kind of graph that can take loops into account, rather than an Attack Tree.

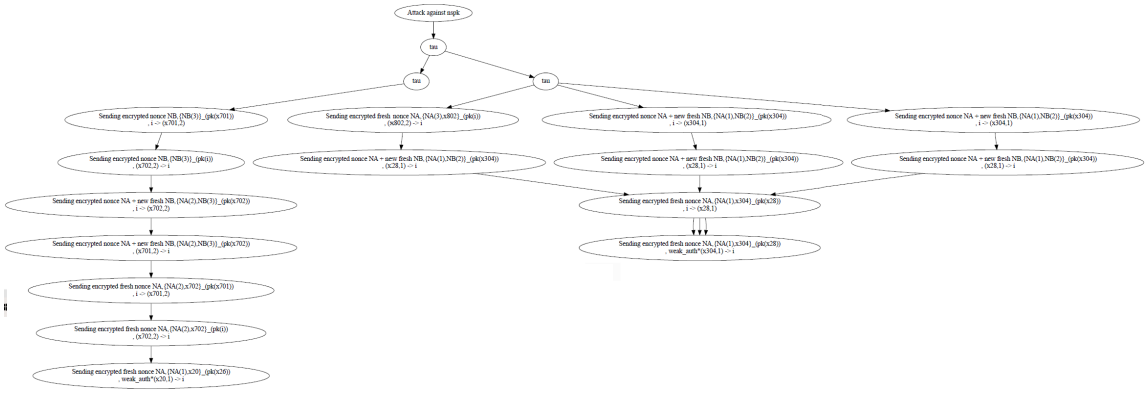

Figure 5.22: Attack Tree about NSPK attack (case 1: content, sender and receiver).

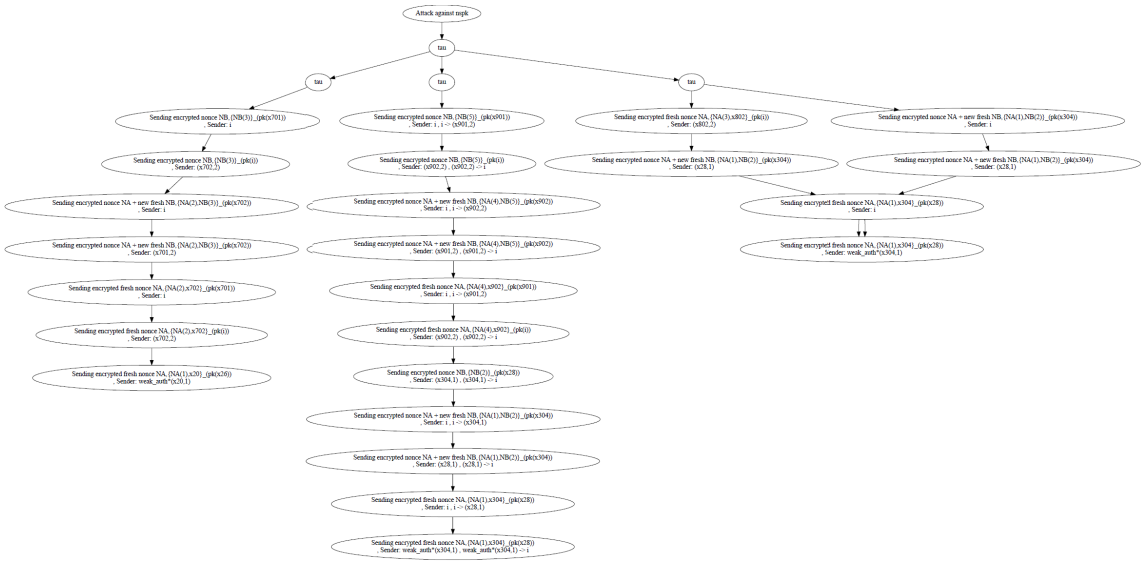

Figure 5.23: Attack Tree about NSPK attack (case 3: content and receiver).

5.2.3 Generation of RisQFLan's code

As a final part, also in this second case study, the equivalent code about the Attack Trees for the RisFQLan tool was generated. The results are comparable to the first example in Figure 5.16, with the exception that the tree in this study's scenario is bigger and slightly more complicated. In order to get a better understanding of the system and any potential threats to it, it is also possible to depict the final trees in this example using the graph-based security framework.

5.3 Third example: another attack trace of multiple attacks

To provide more truthfulness and reliability about our work with the OFMC tool, we chose a third case study that is similar to the previous one. This time OFMC was used to generate an attack trace concerning the Selfi protocol, i.e., an authentication protocol that allows two parts A and B to authenticate themselves on their two nonces N1 and N2 respectively. Figure 5.24 displays the code used as the model checker tool's input with reference to this protocol. As we outlined in the previous case study, nonces are useful because the creator can be sure that the response is not older than the nonce contained in it, in case a third party is required to include the nonce in his response. In the code for the attack trace creation we can note that the message exchange begins with A transmitting to B the following information: A together with B and his generated nonce N1. Afterwards, B answers by transmitting the encrypted secret keys (both keys about A and B) together with A, B and his created nonce N2. The nonces are then excluded from the final exchange because they have already completed their task of ensuring the freshness of the messages sent between the parties. For this reason, in the last message exchange A sends to B the encrypted keys along with A and B. By executing these steps, B can verify the identity of A by using N1, and A can authenticate B as well but using N2.

```
Protocol: Selfi

Types: Agent A,B;
       Number N1,N2;
       Function secretk,h

Knowledge: A: A,B,exp(exp(g,secretk(A)),secretk(B)),h;
           B: A,B,exp(exp(g,secretk(A)),secretk(B)),h;

Actions:

A->B: A,B,N1
B->A: A,B,N2,  h(h(exp(exp(g,secretk(A)),secretk(B)),N1,N2),N1,N2)
A->B: A,B,   h(h(exp(exp(g,secretk(A)),secretk(B)),N1,N2),N1,N2,
               h(h(exp(exp(g,secretk(A)),secretk(B)),N1,N2),N1,N2))

Goals:

B authenticates A on N1
A authenticates B on N2
```

Figure 5.24: OFMC code for the generation of the attack trace about protocol SELFI.

After running the code that was just described and shown in Figure 5.24, we reached an attack trace that outlines multiple attacks to the Selfi protocol. The initial part of the obtained trace is depicted below in Figure 5.25.

```
ATTACK TRACE:
**ALLIN:
########## Attack 1 ##########
weak_auth*(x601,1) -> i: Sends Participants And Nonce N1,x601,x602,N1(1)
(x32,2) -> i: Send Participants And Nonce N1,x32,x34,N1(2)
i -> (x601,2): Send Participants And Nonce N1,x602,x601,N1(1)
(x601,2) -> i: Send Particip And N2 And Encr Key,x602,x601,N2(3),h(h(exp(exp(g,secretk(x602)),secretk(x601)),N1(1),N2(3)),N1(2),N2(3)
i -> (x601,1): Send Particip And N2 And Encr
Key,x601,x602,N2(3),h(h(exp(exp(g,secretk(x601)),secretk(x602)),N1(1),N2(3)),N1(1),N2(3))
(x601,1) -> i: Send Participants And Encr Key,x601,x602,h(h(exp(exp(g,secretk(x601)),secretk(x602)),N1(1),N2(3)),N1(1),N2(3))


########## Attack 2 ##########
weak_auth*(x701,1) -> i: Sends Participants And Nonce N1,x701,x702,N1(1)
(x702,2) -> i: Sends Participants And Nonce N1,x702,i,N1(2)
i -> (x701,2): Sends Participants And Nonce N1,x702,x701,N1(1)
(x701,2) -> i: Send Particip And N2 And Encr
Key,x702,x701,N2(3),h(h(exp(exp(g,secretk(x702)),secretk(x701)),N1(1),N2(3)),N1(1),N2(3))
i -> (x701,1): Send Particip And N2 And Encr
Key,x701,x702,N2(3),h(h(exp(exp(g,secretk(x701)),secretk(x702)),N1(1),N2(3)),N1(1),N2(3))
(x701,1) -> i: Send Participants And Encr Key,x701,x702,h(h(exp(exp(g,secretk(x701)),secretk(x702)),N1(1),N2(3)),N1(1),N2(3))
i -> (x702,2): Send Particip And N2 And Encr Key,x702,i,x606,h(h(exp(exp(g,secretk(x702)),secretk(i)),N1(2),x606),N1(2),x606)
(x702,2) -> i: Send Participants And Encr Key,x702,i,h(h(exp(exp(g,secretk(x702)),secretk(i)),N1(2),x606),N1(2),x606)

########## Attack 3 ##########
weak_auth*(x701,1) -> i: Sends Participants And Nonce N1,x701,x702,N1(1)
(x32,2) -> i: Sends Participants And Nonce N1,x32,i,N1(2)
i -> (x701,2): Sends Participants And Nonce N1,x702,x701,N1(1)
(x701,2) -> i: Send Particip And N2 And Encr
Key,x702,x701,N2(3),h(h(exp(exp(g,secretk(x702)),secretk(x701)),N1(1),N2(3)),N1(1),N2(3))
```

Figure 5.25: Attack trace containing multiple attacks to Selfi protocol.

The obtained attack trace confirms that the protocol is vulnerable to multiple attacks against the same protocol. The potential threats concern an outsider who might compromise the parties' message-exchange authentication. We go on to the next step, in order to obtain the Attack Tree and overcoming the identified threats. As we observed with the prior

case studies, to perform process mining, the attack trace must be turned into a suitable event log.

5.3.1 Translation from attack trace to event log

The attack trace was first transformed into a Pandas Dataframe using the same translation script that was shown in Figure 5.5, just like in the previous example. The resulting Dataframe is displayed below in Figure 5.26. Again, we only included the first four attacks that appear in the trace in order to keep things simple, but it is possible to keep any chosen number. As we said before, we opted to minimize the attacks picked from the trace for the sake of our explanation, because a greater number would result in a wider and more chaotic Attack Tree.

	time:timestamp	concept:activity	@@index	case:concept:info
0	2023-02-01 17:44:40.449960	Sends Participants And Nonce N1,x601,x602,N1(...	0	Sends Participants And Nonce N1
1	2023-02-01 17:46:40.449960	Send Participants And Nonce N1,x32,x34,N1(2)\...	0	Send Participants And Nonce N1
2	2023-02-01 17:48:40.449960	Send Participants And Nonce N1,x602,x601,N1(1...	0	Send Participants And Nonce N1
3	2023-02-01 17:50:40.449960	Send Particip And N2 And Encr Key,x602,x601,N...	0	Send Particip And N2 And Encr Key
4	2023-02-01 17:52:40.449960	Send Particip And N2 And Encr Key,x601,x602,N...	0	Send Particip And N2 And Encr Key
5	2023-02-01 17:54:40.449960	Send Participants And Encr Key,x601,x602,h(h(...	0	Send Participants And Encr Key
6	2023-02-01 17:57:40.449960	Attack against SELFI	0	
7	2023-02-01 17:58:40.449960	Sends Participants And Nonce N1,x701,x702,N1(...	1	Sends Participants And Nonce N1
8	2023-02-01 17:59:40.449960	Sends Participants And Nonce N1,x702,i,N1(2)\...	1	Sends Participants And Nonce N1
9	2023-02-01 18:00:40.449960	Sends Participants And Nonce N1,x702,x701,N1(...	1	Sends Participants And Nonce N1
10	2023-02-01 18:01:40.449960	Send Particip And N2 And Encr Key,x702,x701,N...	1	Send Particip And N2 And Encr Key
11	2023-02-01 18:02:40.449960	Send Particip And N2 And Encr Key,x701,x702,N...	1	Send Particip And N2 And Encr Key
12	2023-02-01 18:03:40.449960	Send Participants And Encr Key,x701,x702,h(h(...	1	Send Participants And Encr Key
13	2023-02-01 18:04:40.449960	Send Particip And N2 And Encr Key,x702,i,x606...	1	Send Particip And N2 And Encr Key
14	2023-02-01 18:05:40.449960	Send Participants And Encr Key,x702,i,h(h(exp...	1	Send Participants And Encr Key
15	2023-02-01 18:07:40.449960	Attack against SELFI	1	
16	2023-02-01 18:08:40.449960	Sends Participants And Nonce N1,x701,x702,N1(...	2	Sends Participants And Nonce N1
17	2023-02-01 18:09:40.449960	Sends Participants And Nonce N1,x32,i,N1(2)\n...	2	Sends Participants And Nonce N1
18	2023-02-01 18:10:40.449960	Sends Participants And Nonce N1,x702,x701,N1(...	2	Sends Participants And Nonce N1
19	2023-02-01 18:11:40.449960	Send Particip And N2 And Encr Key,x702,x701,N...	2	Send Particip And N2 And Encr Key

Figure 5.26: Dataframe containing log entries about attacks against Selfi protocol.

In this instance as well, the PM4PY library was used to convert the Dataframe into a XES event log. The code is equivalent to the one we illustrated and described in Figure 5.8. Again, the relevant data required from the log is: ID, timestamp, activity label and information. The generated XES file can now be used as input for the Process Mining technique in the following phase.

5.3.2 From event log to process tree and attack tree

As we stated before in Figure 5.9, we can go on with the process by utilizing the PM4PY functions `view_process_tree` and `discover_process_tree_inductive` By carrying out this, using the event log as input, we may learn about and see the system's linked Process Tree. Again , we distinguished three different possibilities for the 'activity' field shown in the nodes of the tree: first, the activity could include the sender, the recipient, and the content of the message; second, it could include include only the sender and receiver; and third, it could include only the sender and content. We were able to produce three different Process Trees as a result. Due to space limitations, we opted to omit the first and third Process Trees, leaving only the second one, depicted in Figures 5.27 , which focuses only on sender and receiver.

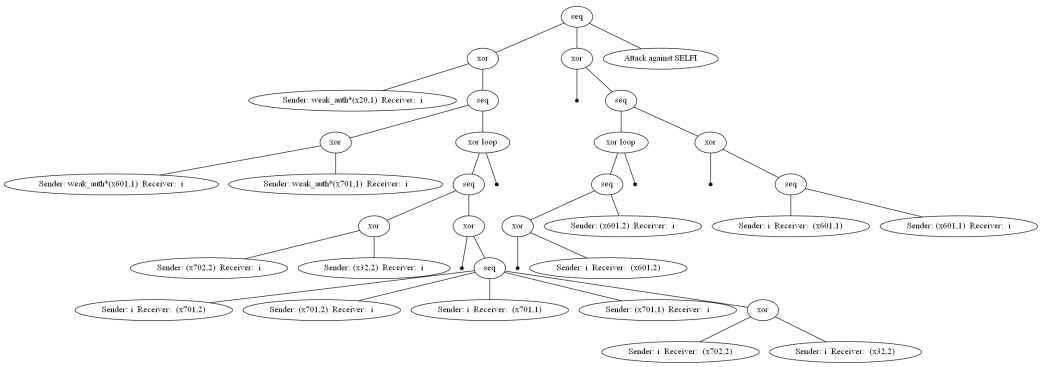

Figure 5.27: Process Tree about Selfi attack (case 2: content and sender).

Also in this example, Figure 5.27 shows that the Process Tree about case 2 contains some 'xor loops' operators that cannot be converted into Attack Tree operators for the same reasons stated in the previous subsection. Because of this, in order to generate only two Attack Trees, namely the one with nodes containing content, sender, and receiver, and the one about content and sender, we only applied our translation rules to the first and third Process Trees. The obtained ATs are shown in Figures 5.28 and 5.29 respectively.

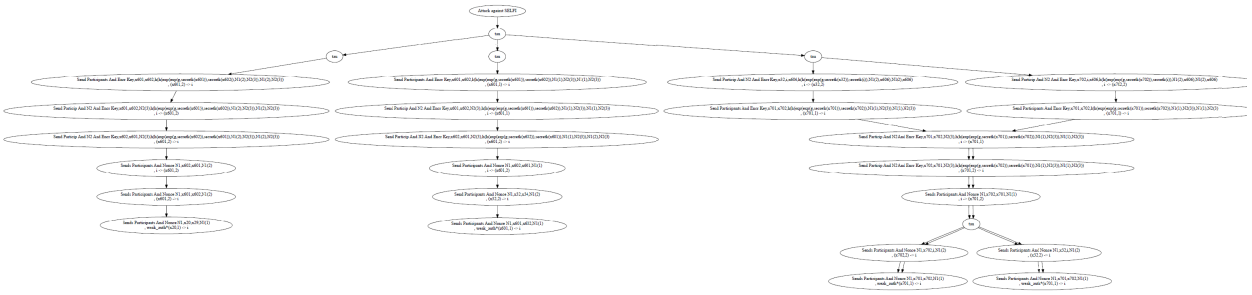

Figure 5.28: Attack Tree about Selfi attack (case 1: content, sender and receiver).

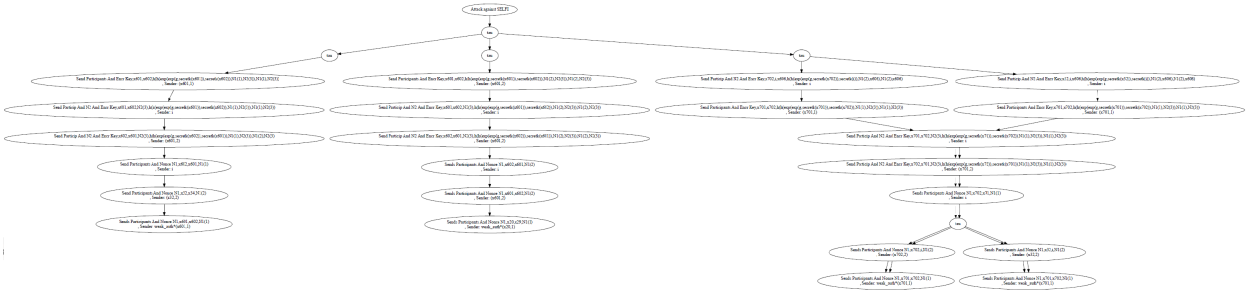

Figure 5.29: Attack Tree about Selfi attack (case 3: content and receiver).

5.3.3 Generation of RisQFLan's code

Concluding this third case study, also here the RisFQLan codes were generated, starting from the obtained XML Attack Trees. What we achieved looks like the previous examples (Figure 5.16), and also this case study generated wider and slightly more complicated trees with respect to the first case. Essentially, also in this instance it is possible to convert the final Attack Tree. This allows one to use the RisQFLan graph-based security framework, to achieve a better overview of the analyzed system and to identify possible threats that could show up.

5.4 A 'bad' example: malware infection logs

For the scope of our work we provide also another example in which real event logs are used without the need of OFMC. To be precise, these logs are about malware infection traffic and they are provided by Strato-

sphere IPS [6].The Stratosphere IPS feeds itself with models created from real malware traffic captures. Their aim is to continuously monitor the threat landscape for new emerging threats, retrieving malicious samples to capture the traffic. The selected project is named 'Malware Capture Facility Project' and it is responsible for making the long-term malware captures. This dataset of traffic logs provide a real scenario in which a machine is not infected, then infected with a malware and after some time this is cleaned up. Specially in network computer security it is really important to have good datasets, because the data in the networks is infinite, changing, varied and with a high concept drift. For this reason, this data traffic is about malware but also about normal activities. The specific capture taken into account is about a normal computer working with a real user for some time, then a malware infection happens, finally it is cleaned and the user continues working normally for some time.

5.4.1 From real traffic logs to activity logs

Our initial aim is to translate this traffic log dataset into a proper activity log suitable for the process mining algorithm. Note that in this case we do not use the OFMC tool because we already have some traffic logs and therefore we do not need to start from an attack trace generation. Firstly, the algorithm requires some logs in XES format and secondly we have to pass the right columns of data needed in the process mining procedure. As previously introduced, the specific columns of information that we want in the final XES log are the following:

- **ID**: progressive numbers so that the different activities can be distinguished between each others.
- **Timestamp**: date and time of the single line of log. For this specific traffic log a translation of the timestamps is needed: in the original file they are stored as total amount of seconds, while we need a real date and time for the purpose of the process mining.
- **Activity**: it illustrates the typical information present in the original

traffic log. In this column the IP of the source, the IP of the destination, the used protocol and the length are listed.

- **Information**: brief description of a specific atomic activity shown in a single line of the logs.

The first step is accomplished through Pandas Python's library. It allows the translation of the original traffic logs depicted in Figure 5.32 into activity logs suitable for the Process Mining. This transformation is shown in our translation script below in Figure 5.31. To begin with, the initial traffic log in CSV format provided by the repository is imported and converted into a Dataframe using the `read_csv` function from Pandas. The first problem is that in the obtained Dataframe we do not have the proper columns of data that Process Mining requires from an event log. As we saw before, the right structure that we choose for our columns is the following: `timestamp`, `activity`, `index`, `info`. To achieve this, the conversion of the Dataframe into a List of Lists was done using the `tolist()` function. This operation is helpful because now we can loop the List so that we can obtain the strings of data needed and put them in the rows of a new Dataframe, i.e. `updatedPD`, that is the final one. Note that, for what concerns the time, in the initial traffic logs we have amount of seconds, while in the final log we need a proper date and time. To overcome this, we imported `datetime` to set the starting date and `timedelta` to add the seconds to an actual date, and convert everything into a proper date-time format. These translation steps are also summarized in Figure 5.30.

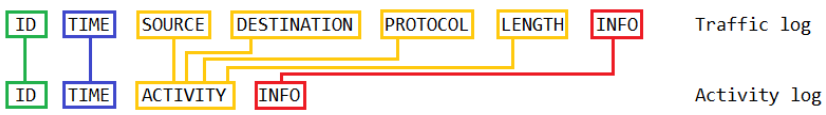

Figure 5.30: From real traffic logs to activity logs.

```
# I read the initial CSV with the logs about the system BEFORE malware
df = pd.read_csv("Documents\Online malware dataset log\BeforeInfection.csv")
listaDF = df.values.tolist()

# Initial header about the first CSV with all the logs
headerRows= ['No','Time','Source','Destination','Protocol','Length','Info']

print("Dataframe with right portion of data but with original header: \n")
print(df)

# starting date and time according to the repository
startingDateTime = datetime(2015, 7, 25, 17, 51, 12)
updatedRows = [] # needed rows structure: timestamp, activity, index, info

# I create the new dataframe with the right columns and the right portion of
    data in which I am interested
for data in listaDF:
    for singleData in data:
        res = singleData.split('"', 1)
        res1 = res[1] # time, source, destination, protocol, length, info
        res2 = res1.split('"', 1)
        timeVariable = (res2[0])  # ONLY TIME
        realTimeVariable = startingDateTime + timedelta(seconds = int(
            timeVariable.split('.',1)[0])) # TIME IN DATE FORMAT
        res3 = res1.split(',', 1)[1] # souce, dest, procotol, lenght, info
        res4 = res3.split(',', 3)[3] length and info
        info = res4.split(',',1)[1] # INFO
        # ACTIVITY (SOURCE, DESTINATION, PROCOTOL AND LENGTH)
        activity = res3.split(',', 7)[0] + "," + res3.split(',', 6)[1] + "," +
             res3.split(',', 6)[2] + "," + res3.split(',', 6)[3]
        singleRow = [realTimeVariable, activity, 0, info]
        updatedRows.append(singleRow)

updatedPD = pd.DataFrame(updatedRows, columns = ['time:timestamp','concept:
    activity','@@index','case:concept:info'])

print("\nNew dataframe with proper columns: ")
print(updatedPD)
```

Figure 5.31: Translation from real traffic logs to activity logs suitable for Process Mining.

As we can see in Figure 5.33, with the translation script the information needed for Process Mining are now together in specific columns and, as a result, a final DataFrame is obtained. This data will be used by the Inductive Miner algorithm provided by the PM4PY library.

	Unnamed: 0	No	Time	Source	Destination	Protocol	Length	Info
0	0	541001,"74765.734239","10.0.0.45","157.56.193....	NaN	NaN	NaN	NaN	NaN	NaN
1	1	541002,"74766.559862","91.190.218.59","10.0.0....	NaN	NaN	NaN	NaN	NaN	NaN
2	2	541003,"74766.596330","91.190.218.59","10.0.0....	NaN	NaN	NaN	NaN	NaN	NaN
3	3	541004,"74766.608628","10.0.0.45","91.190.218....	NaN	NaN	NaN	NaN	NaN	NaN
4	4	541005,"74770.342714","10.0.0.45","10.0.0.255"...	NaN	NaN	NaN	NaN	NaN	NaN
5	5	541006,"74770.342744","10.0.0.45","10.0.0.255"...	NaN	NaN	NaN	NaN	NaN	NaN
6	6	541007,"74772.171192","10.0.0.45","8.8.8.8","D...	NaN	NaN	NaN	NaN	NaN	NaN
7	7	541008,"74772.306489","8.8.8.8","10.0.0.45","D...	NaN	NaN	NaN	NaN	NaN	NaN
8	8	541009,"74772.307734","10.0.0.45","8.8.8.8","D...	NaN	NaN	NaN	NaN	NaN	NaN
9	9	541010,"74772.479682","8.8.8.8","10.0.0.45","D...	NaN	NaN	NaN	NaN	NaN	NaN
10	10	541011,"74772.480882","10.0.0.45","168.63.15.1...	NaN	NaN	NaN	NaN	NaN	NaN
11	11	541012,"74772.623390","168.63.15.132","10.0.0....	NaN	NaN	NaN	NaN	NaN	NaN
12	12	541013,"74772.623804","10.0.0.45","168.63.15.1...	NaN	NaN	NaN	NaN	NaN	NaN
13	13	541014,"74772.624694","10.0.0.45","168.63.15.1...	NaN	NaN	NaN	NaN	NaN	NaN
14	14	541015,"74772.762938","168.63.15.132","10.0.0....	NaN	NaN	NaN	NaN	NaN	NaN
15	15	541016,"74773.659543","10.0.0.45","195.50.88.1...	NaN	NaN	NaN	NaN	NaN	NaN

Figure 5.32: Initial table format of real traffic logs.

	time:timestamp	concept:activity	@@index	case:concept:info
0	2015-07-26 14:37:17	"10.0.0.45","157.56.193.33","TCP","54"	0	"[TCP Dup ACK 540819#1] 49711 > 443 [ACK] Se...
1	2015-07-26 14:37:18	"91.190.218.59","10.0.0.45","TCP","54"	1	"[TCP Keep-Alive] 12350 > 49703 [ACK] Seq=37...
2	2015-07-26 14:37:18	"91.190.218.59","10.0.0.45","TCP","54"	2	"12350 > 49703 [ACK] Seq=3789 Ack=8930 Win=1...
3	2015-07-26 14:37:18	"10.0.0.45","91.190.218.59","TCP","54"	3	"[TCP Dup ACK 540236#2] 49703 > 12350 [ACK] ...
4	2015-07-26 14:37:22	"10.0.0.45","10.0.0.255","DB-LSP-DISC/JSON","146"	4	"Dropbox LAN sync Discovery Protocol, JavaScri...
5	2015-07-26 14:37:22	"10.0.0.45","10.0.0.255","DB-LSP-DISC/JSON","146"	5	"Dropbox LAN sync Discovery Protocol, JavaScri...
6	2015-07-26 14:37:24	"10.0.0.45","8.8.8.8","DNS","78"	6	"Standard query 0x7302 A api.trap.skype.net"\t...
7	2015-07-26 14:37:24	"8.8.8.8","10.0.0.45","DNS","164"	7	"Standard query response 0x7302 A api.trap.sky...
8	2015-07-26 14:37:24	"10.0.0.45","8.8.8.8","DNS","78"	8	"Standard query 0x5df5 AAAA api.trap.skype.net...
9	2015-07-26 14:37:24	"8.8.8.8","10.0.0.45","DNS","148"	9	"Standard query response 0x5df5 AAAA api.trap....
10	2015-07-26 14:37:24	"10.0.0.45","168.63.15.132","TCP","66"	10	"56565 > 443 [SYN] Seq=0 Win=8192 Len=0 MSS=...
11	2015-07-26 14:37:24	"168.63.15.132","10.0.0.45","TCP","66"	11	"443 > 56565 [SYN, ACK] Seq=0 Ack=1 Win=8192...
12	2015-07-26 14:37:24	"10.0.0.45","168.63.15.132","TCP","54"	12	"56565 > 443 [ACK] Seq=1 Ack=1 Win=66240 Len...
13	2015-07-26 14:37:24	"10.0.0.45","168.63.15.132","SSLv2","99"	13	"Client Hello"\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t...
14	2015-07-26 14:37:24	"168.63.15.132","10.0.0.45","TCP","54"	14	"443 > 56565 [RST, ACK] Seq=1 Ack=46 Win=0 L...
15	2015-07-26 14:37:25	"10.0.0.45","195.50.88.19","TCP","54"	15	"56559 > 80 [FIN, ACK] Seq=1 Ack=1 Win=66560...

Figure 5.33: Final table of activity logs suitable for Process Mining obtained from the translation script.

5.4.2 From activity log to Process Tree

After obtaining the event log DataFrame about the whole system, it is necessary to export the data in the XES format. For this specific case it was considered appropriate to split the obtained event log into two parts: one about the system's behaviour before the infection and another one concerning the post-malware. In this case this step is allowed because the analyzed log repository provides the exact time and date in which the in-

fection takes place. This preliminary operation could help us notice which are the differences between a system that is infected and another one that is not.
The next step is to apply the usual Process Mining procedure outlined in Section 2.2 using PM4PY library from Python, both for the data before and after the infection. The two obtained Process Trees are depicted in Figure 5.34.

Figure 5.34: Process Trees respectively about before and after malware infection.

5.4.3 The problem: Attack Tree generation

At this point we reach a problem concerning this specific case study. As we can see in the two obtained Process Trees in Figure 5.34, a difference in the behaviour of the system cannot be noticed just looking at what is happening into the Process Trees or into the event logs. To be precise, in this example it is not possible to spot which are the steps of the attacker that lead him to the goal of the attack, i.e. the malware infection. To make this tool work properly, some activities or atomic steps about the attacker must be detected starting from the analyzed event log, so that they can be depicted in the generated Process Tree. For these reasons we tried our tool on other examples, i.e. the previous case studies, and for the scope of this thesis we decided to use also an automated protocol verification tool, i.e. OFMC, that allows one to generate attack traces from a given protocol in input. From the attack traces we showed how it can be possible to obtain event logs as input for our tool. However, many additional case studies may focus on conventional activity logs produced by normal systems similar to the one just examined. Indeed, many others event logs can be examined and can generate Attack Trees with the help of our tool, without the usage of OFMC.

5.5 Contribution w.r.t. related works

The purpose of this section is to discuss the reasons behind the design of the tool that was introduced in this thesis. To do this, we updated the initial scheme (Figure 1.6) about the related papers and associated weaknesses that was discussed in the 'related work' part (Section 1.4). Basically we added one new column on the right, namely the one that refers to our tool. The updated version is now represented below in Figure 5.35. To be more precise, we point out that our tool does not require complicated formal engines or huge databases of typical attacks to work. In fact, what we require is either a protocol to be provided as input to the OFMC tool in order to build a log from the trace, or an event log from a real system that

can represents some of the attacker's actions. Additionally, the emphasis is solely on Attack Trees and not on other kinds of graphs. By concentrating on a single type of graph, we can better understand its advantages and drawbacks, as well as propose alternatives based on the disadvantages of the Attack Tree. Finally, we can say that we developed a fully automated tool that includes the code and does not just analyze socio-technical systems; rather, we take a broader variety of possibilities into account in our case studies.

Weaknesses	Philips et. al.	Lallie et. al.	Wojciech et. al.	Kordy et. al.	Sheyner et. al.	Pinchinat et. al.	Ibrahim et. al.	David et. al.	Automatization tool
Need DB of typical attacks	✓								
Automatization not taken into account		✓							
Sophisticated formal engines			✓						
Focus on other type of attack graph				✓			✓		
Not scalable	✓				✓				
Code not available	✓	✓			✓				
Not entirely automated						✓			
Only about socio-technical systems								✓	

Related papers

Figure 5.35: Related papers and weaknesses + the presented tool.

However, even though the tool has many advantages, while developing and analyzing our case studies it revealed some considerable drawbacks related to some instances. These issues will be also discussed and addressed in the final chapter of the thesis. Just before we made clear that for the tool to function effectively and produce a meaningful Attack Tree, some visible attacker's steps must be documented in the log. This prerequisite is not trivial, because it is unlikely that the attacker will not take all the reasonable precautions to hide his activities. Another important precondition is that we require a labeled log in order to have some tags for the nodes shown in the final Attack Tree. Unfortunately, the fact that various systems do not automatically label the logs makes this situation complicated. Finally, we highlighted that the Attack Tree does not natu-

rally support loops. Thus, if we find this operator in the obtained Process Tree, we will not be able to convert it into an Attack Tree. However, as we said before, chapter 7 will discuss and address the discussed challenges related to our tool.

Chapter 6

Validation

As a way of proving the soundness and validity of this tool, we examined the two main translation processes separately. Figure 6.1 shows the overall idea of this approach. Our validation consists in two steps: the first phase (in blue) in which the event logs are converted into a Process Tree, and verified; and the second phase (in pink) in which the Process Tree is transformed into an Attack Tree and then validated.

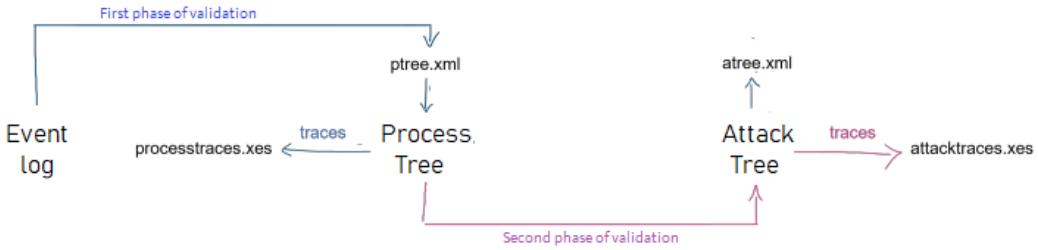

Figure 6.1: Overview of the stages for the validation.

6.1 Validation from event logs to process tree

We used the PM4PY Python module to generate the Process Tree starting from the system's event log. It provides the most famous algorithm implementations in addition to helpful functions connected to process mining. To get the Process Tree associated with the input log, we utilized the `discover_process_tree_inductive` function. As introduced in Section 2.3.1, The Inductive Miner algorithm that we employed has under-

gone extensive testing and consolidation, as described in Section 2, and a sizable body of literature is already available. [3] represents a good contribution to the truthfulness and reliability of the Inductive Miner procedure, moreover all the steps are soundly illustrated and verified.
The same applies to the aforementioned Python library: in [17] the official documentation of PM4PY is provided. As a result, one can gain a better understanding of all the available features and functions that have been extensively demonstrated and tested in the existing literature.

6.2 Validation from process tree to attack tree

Now that the PM4PY Python library has been used, we have a Process Tree. As an XML file, the PT constitute the input for the translation rules that generate the final Attack Tree. Then, as a final step in the verification process, a collection of traces was created for each example taken into consideration, starting with one model. The aim was to see if the translated model could make use of the traces as well. Hence, starting from the Process Tree, it was possible to generate traces and save them as event log. A replay algorithm was then used to determine whether the traces were correctly convertible to the translated model. This was done translating the initial model into the other, and finally the traces were obtained from the initial one. This will authenticate the final stage of the whole process by demonstrating that the latter replicate the identical actions as the former. As a result, the Process Trees were translated into Attack trees, and both the models managed to execute the same traces. Figure 6.2 depicts the translation from Process Tree to Attack Tree: on the left the PT is shown, while on the right we have the related AT.

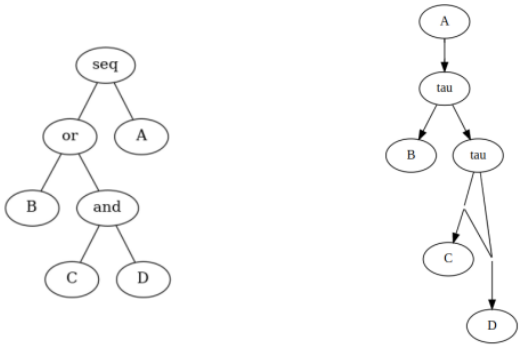

Figure 6.2: From Process Tree to Attack Tree.

Throughout this thesis, we generated traces for each considered example to ensure the validity of our work. Hereinafter we summarize the obtained results for each case study:

- Dummy log generation: from the PT we obtained 4999 traces, which are all successfully replayed by the derived Attack Tree.
- OFMC attack trace about single attack: the Process Tree in this case generated 2999 traces that have been effectively reproduced by the AT.
- First OFMC attack trace about multiple attacks: for this example 9611 traces were obtained and subsequently replicated by the Attack Tree.
- Second OFMC attack trace about multiple attacks: also in this case study the derived Attack Tree successfully replayed all 7637 traces obtained from the Process Tree.
- Malware infection logs: in this last example we have 45410 and 180274 traces related to the before and after malware respectively.

To sum up, the validation process entailed examining the soundness of each stage of translation, from the event logs to the final Attack Tree. This could guarantee the reliability of each transformation. To do this, we divided the validation process into two stages, and after that, we verified our tool.

Chapter 7

Conclusion and Future Works

In conclusion, the aim of this thesis was to introduce an automatic tool for generating Attack Trees. They can be seen as graphical representations of all the potential attacks to a system. The generation can be done starting from dummy, i.e. artificially generated, or real event logs. We also showed in our case studies that it is possible to use the model checker tool OFMC to obtain an attack trace regarding a specific protocol, and then translate it into an event log. Using the Inductive Miner algorithm and a Python library we can then generate a Process Tree of the system starting from the logs, both from original or translated ones. Using Process Trees, it is possible to depict all the information collected by a particular system and understand what is happened in the past. Following some translation rules, the Process Tree is transformed into the related XML Attack Tree. The tree was then converted into a format suitable for a graph-based quantitative security risk framework called RisQFLan, using a translation script that was written as a last step. The decision to include also RisQFLan stems from the fact that it combines various well known rigorous notions and methods in an Eclipse-based domain specific framework. This can help to visualize and better understand the overall situation of the system.
As a future implementation, also Attack-Defense Trees, which consider both attacks and countermeasures, could be taken into account starting from a selected event log. This would be an interesting extension of the tool. It would make it more complete and informative for the security an-

alyst that want to protect the security of a system, providing also specific protection assets. Future integrations may also focus on finding a solution to the issue of constructing the Attack Tree in the presence of xor loops in the Process Tree, e.g. by producing a different type of graph that handles loops. Another future improvement could be to overcome the issue regarding the last use case. In this way, the tool could automatically generate Attack Trees starting from a more generic and less precise event log, without the need for the attacker's steps to be reported in the starting log. One last interesting extension could be the use of Machine Learning for log labeling. By doing this, the tool would not be limited solely to labeled logs, but it could also include those that are not.

To conclude, in the present work a way for the automatic generation of Attack Trees has been introduced, starting from labeled event logs or also from attack traces generated by the model checker OFMC. The tool has interesting opportunities to be extended and improved. This would further increase its features and would make it useful for many more different cases. In this way, the tool might become even more frequently usable in the real world and might support companies by helping them strengthen their defenses against cyber threats.

Bibliography

[1] B. Schneier (1999), *Attack Trees*, Dr. Dobb's journal.

[2] Maurice ter Beek, A. Lluch Lafuente, A. Legay (2021), *Quantitative Security Risk Modeling and Analysis with RisQFLan*, Computers Security 109.

[3] Wil van der Aalst (2016) *Process Mining*, Springer, Data Science in Action.

[4] S. Mauw AND M. Oostdijk (2005), *Foundations of Attack Trees*, International Conference on Information Security and Cryptology.

[5] B. Kordy, L. Piètre-Cambacédès, P. Schweitzer (2014), *Dag-based attack and defense modeling: Don't miss the forest for the attack trees*, Computer Science Review, vol. 13-14.

[6] *Stratosphere IPS Malware Capture Facility Project*, `https://www.stratosphereips.org/datasets-overview/`.

[7] L. Zhang (2012), *The Research of Log-Based Network Monitoring System*, Springer, Advances in Intelligent Systems.

[8] T. Sipola, A. Juvonen, J. Lehtonen (2011), *Anomaly Detection from Network Logs Using Diffusion Maps*, Springer, Engineering Applications of Neur.

[9] Jakub Breier and Jana Branišová (2015), *Anomaly detection from log files using data mining techniques*, Springer, Information Science and Applications.

[10] Min Du, Feifei Li, Guineng Zheng (2017), *Deeplog: Anomaly detection and diagnosis from system logs through deep learning*, Springer, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.

[11] Xiaoyun Li, Pengfei Chen, Linxiao Jing (2020), *SwissLog: Robust and Unified Deep Learning Based Log Anomaly Detection for Diverse Faults*, IEEE, Information Science and Applications.

[12] Xiaoyun Li, Pengfei Chen, Linxiao Jing (2020), *Unsupervised log message anomaly detection*, IEEE, Amir Farzad and T Aaron Gulliver.

[13] J. Zhu, S. He, J. Liu (2019), *Tools and Benchmarks for Automated Log Parsing*, ICSE-SEIP, International Conference on Software Engineering: Software Engineering in Practice.

[14] Mell P., Hu V., Lippmann R. (2003), *An overview of issues in testing intrusion detection systems*, National Insititute of Standards and Technology, Technical Report NIST IR 7007.

[15] A. C¸ amtepe and B. Yener (2007), *Modeling and detection of complex attacks*, IEEE, Proceedings of the 3rd International Conference on Security and Privacy in Communication Networks (SecureComm'07).

[16] G. D. Federico, A. M. Konsta (2022), *Process Trees to Attack Trees: Do Not Miss the Trees*.

[17] *PM4PY Python package documentation*, `https://pm4py.fit.fraunhofer.de/docs`.

[18] *OFMC download page*, `www.avantssar.eu`

[19] *AVISPA Deliverable*, `www.avispa-project.org`

[20] *AVISPA Library of Protocols*, `ls.http://www.avispa-project.org/library`

[21] S. J. Leemans, D. Fahland, and W. M. Van Der Aalst (2013), *Discovering block-structured process models from event logs-a constructive approach*, Springer, International conference on applications and theory of Petri nets and concurrency, pp. 311–32.

[22] L. Painton Swiler, C. Phillips (1998), *A graph-based system for network-vulnerability analysis*, Proceedings of the 1998 Workshop on New Security Paradigms

[23] J. Bal H., S. Lallie, K. Debattista (2020), *A review of attack graph and attack tree visual syntax in cyber security*, Computer Science Review

[24] B Fila, W. Wideł, M. Audinot (2014), *Beyond 2014: Formal methods for attack tree–based security modeling*, ACM Comput. Surv., 52(4)

[25] M. Bozga, A. David, A. Hartmanns (2012), *State-of-the-Art Tools and Techniques for Quantitative Modeling and Analysis of Embedded Systems*, EDAA, Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12).

[26] P. Schweitzer, B. Kordy, L. Pietre-Cambac (2014), *Dag-based attack and defense modeling: Don't miss the forest for the attack trees*, Computer Science Review, 13-14:1–38

[27] S. Jha, O. Sheyner, J. Haines (2002), *Automated generation and analysis of attack graphs*, Symposium on Security and Privacy

[28] G. Agha and K. Palmskog (2018), *A survey of statistical model checking*, ACM Trans. Model. Comp. Simul., vol. 28, no. 1.

[29] A. Legay, A. Lukina, L. Traonouez (2019), *Statistical Model Checking*, Springer, Computing and Software Science: State of the Art and Perspectives, vol. 10000.

[30] Maurice ter Beek, A. Lluch Lafuente, A. Legay (2020), *A framework for quantitative modeling and analysis of highly (re)configurable systems*, IEEE, Trans. Softw. Eng., vol. 46, no. 3.

[31] Kordy B., Mauw S., Radomirovic (2014), *Attack–Defense Trees*, Journal of Logic and Computation.

[32] D Vojtisek, S. Pinchinat, M. Acher (2016), *Atsyra: An integrated environment for synthesizing attack trees*, Graphical Models for Security

[33] A. Alsheikh, M Ibrahim (2019), *Automatic hybrid attack graph (AHAG) generation for complex engineering systems*

[34] R. Rydhof, N. David, A. David (2015), *Modelling social-technical attacks with timed automata*, International Workshop on Managing Insider Security Threats

[35] Vigo R., Nielson F., Nielson (2014), *Automated Generation of Attack Trees*, IEEE, Computer Security Foundations Symposium (CSF).

[36] Kordy B., Mauw S., Pieters W. (2014), *Towards Automating the Construction Maintenance of Attack Trees: a Feasibility Study*, GraMSec. EPTCS.

[37] W. Wideł, M. Audinot, B. Fila (2019), *Beyond 2014: Formal methods for attack tree–based security modeling*, ACM Comput. Surv., vol. 52.

[38] A. Weijters, W. M. van Der Aalst, A. A. De Medeiros (2017), *Process mining with the heuristics miner-algorithm*, Technische Universiteit Eindhoven, Tech. Rep. WP, vol. 166.

[39] Sebastian Modersheim and Luca Viganò (2009), *The Open-source Fixed-point Model Checker for Symbolic Analysis of Security Protocols*, Springer, Foundations of Security Analysis and Design.

[40] S. J. Leemans, D. Fahland, W. M. Van Der Aalst (2013), *Discovering block-structured process models from event logs-a constructive approach*, Springer, International conference on applications and theory of Petri nets and concurrency.

[41] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst (2013), *Discovering Block-Structured Process Models from Event Logs Containing Infrequent*

Behaviour, Springer, International Workshop on Business Process Intelligence.

[42] M. G. Ivanova, C. W. Probst, R.Hansen (2015), *Attack Tree Generation by Policy Invalidation*, Springer, IFIP International Conference on Information Security Theory and Practice.

List of Figures

Survey: Automatic Generation of Attack Trees and Attack Graphs

Alyzia-Maria Konsta
DTU Compute
Kongens Lyngby, Denmark
akon@dtu.dk

Beatrice Spiga
DTU Compute
Kongens Lyngby, Denmark
s225953@student.dtu.dk

Alberto Lluch Lafuente
DTU Compute
Kongens Lyngby, Denmark
albl@dtu.dk

Nicola Dragoni
DTU Compute
Kongens Lyngby, Denmark
ndra@dtu.dk

***Abstract*—Graphical security models constitute a well-known, user-friendly way to represent the security of a system. These kinds of models are used by security experts to identify vulnerabilities and assess the security of a system. The manual construction of these models can be tedious, especially for large enterprises. Consequently, the research community is trying to address this issue by proposing methods for the automatic generation of such systems. In this work, we present a survey illustrating the current status of the automatic generation of two kinds of graphical security models -Attack Trees and Attack Graphs. The goal of this survey is to present the current methodologies used in the field, compare them and present the challenges and future directions for the research community.**

***Index Terms*—Automatic Generation, Attack Trees, Attack Graphs, Survey**

I. Introduction

During the last few decades, the use of electronic devices has spread significantly. Companies and individuals are using technology for both personal and work-related reasons. As a consequence, a huge amount of personal and sensitive data is stored or processed on computer networks. The research community has focused on finding ways to protect this data from malicious actors.

One well-known solution for assessing the security of a system is the graphical security models. These models represent security scenarios and help security experts identify the system's weaknesses. Their graphical representation constitutes a user-friendly way to analyze the security of the system. In the scope of this paper, we are going to examine only two kinds of these graphical representations -Attack Trees and Attack Graphs. For a comprehensive overview of all the available formalism, we suggest the reader refer to [15].

Security experts manually produce the graphical security models. This procedure, especially for large systems, can be tedious, error-prone, and non-exhaustive. Consequently, the need for automatic procedures arose and the research community has focused on addressing this issue.

In this work, we present an exhaustive survey on the automatic generation of Attack Trees and Attack Graphs. The main goal of this work is to provide an overview of the field and to identify current challenges and future research opportunities. The survey presents:

- state of the art on the automatic generation of Attack Trees and Attack Graphs.
- a quantitative study of the works included in these survey and some important features to offer an overview of the field.
- a classification of the works included in this survey based on the methodologies used in every work to automatically generate the Attack Trees/Graphs.
- a comparative study based on the main characteristics of every work.
- limitations and future directions for the research community.

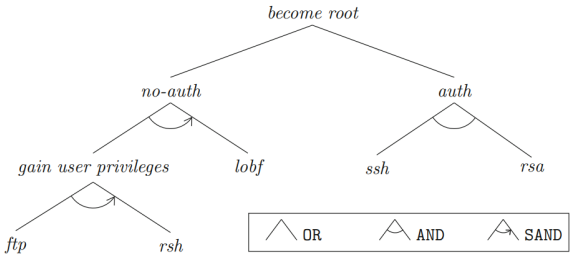

Fig. 1. Attack Tree [12]

The paper is structured as follows: Section II presents the main concepts of Attack Trees and Attack Graphs, as well as an overview to the categories used to classify the papers under study, Section III presents the research method used to structure our research, Section IV presents the related work, Section V presents the quantitative study and the overview of the categories, Section VI presents the classification of the papers, Section VII discusses the challenges and proposes future research directions and Section VIII concludes the paper.

II. Background

In this section, we are going to present the basic concepts used in this survey.

A. Attack Trees

An Attack Tree is a graphical representation of potential attacks to the system's security in the form of a tree. Initially introduced by Schneiner [25], this type of representation enables developers to identify the vulnerabilities of the system

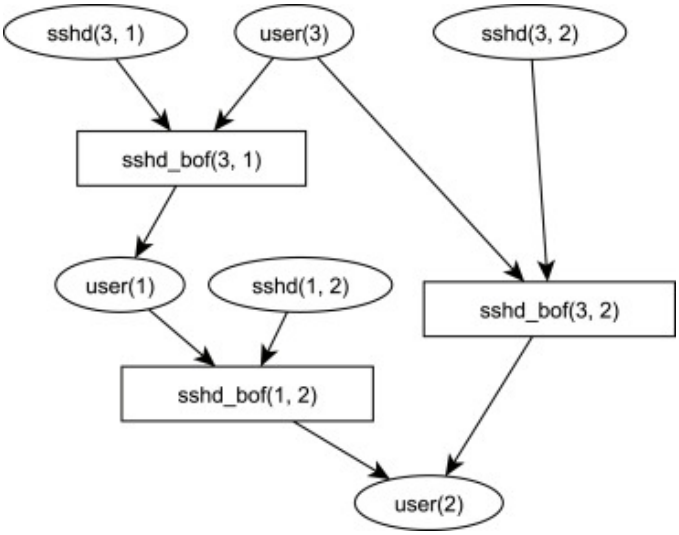

Fig. 2. Attack Graph [ALB: from] [17] [ALB: Resolution is bad. Use a PDF.]

and facilitates the implementation of countermeasures. They are modeled attack scenarios presented in a hierarchical way with each labeled node corresponding to a sub-goal of the attacker and the **root** being the main one - the global goal of the attack. The rest of the labeled nodes can be either **children of a node**, a refinement of the the parent's goal into subsidiary goals, or **leaf nodes**, representing attacks that cannot be further refined in order to be implemented, also called basic actions. The basic formal model of Attack Trees incorporate two types of refinements: `OR` and `AND`. `OR` nodes represent disjunction (choice) where the parent node's goal is achieved when at least one the children's sub-goals is achieved, `AND` nodes represent conjunction (aggregation) thus requiring for all children's sub-goals to be fulfilled. An additional refinement, relating to the latter, is the `SAND` node, that pose the additional condition for sequential realization [18], [31]. One example of an Attack Tree is illustrated in Figure 1. In summary, there are two separate ways for the attacker to accomplish his goal, namely becoming a root: with or without authentication. The two refined authentication options are ssh and rsa, and both must be used because an AND arc is shown. On the other hand, if no authentication is carried out, the user must first be granted access using ftp and then rsh. Following the acquisition of privileges, lobf comes next. It is readily apparent that the tree notation is very appealing and convenient for a threat analysis process since it can include multiple attacks derived from physical, technical and even human vulnerabilities [31]. In view of this, an Attack Tree can be defined as exhaustive, if it encompass all possible attacks, or succinct, if it covers only network-related exploitation of the system by the attacker [20].

B. Attack Graphs

Attack graphs are graphical representations of all the paths through a system that end in a condition in which an attacker has successfully achieved his malicious goal. They outline all potential vulnerabilities and all possible attack paths in a given network and they are frequently used to represent complex attacks which have multiple paths and goals [17]. Attack Graphs play an important role in network security, as they directly show the existence of vulnerabilities in network and how attackers use these vulnerabilities to implement an effective attack [33].

An Attack Graph or AG is a tuple $G = (S, \tau, S_0, S_s)$, where S is a set of states, $\tau \subseteq S \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states, and $S_s \subseteq S$ is a set of success states [26]. Intuitively, S_s denotes the set of states where the intruder has achieved his goals. Unless stated otherwise, we assume that the transition relation τ is total. We define an execution fragment as a finite sequence of states $s_0, s_1, ..., s_n$ such that $(s_i, s_i + 1) \in \tau$ for all $0 \leq i \leq n$. An execution fragment with $s_0 \in S_0$ is an execution, and an execution whose final state is in S_s is an attack, i.e., the execution corresponds to a sequence of atomic attacks leading to the intruder's goal [26]. One example of an Attack Graph can be seen in Figure 2.

C. Attack Trees vs Attack Graphs

Attack graphs and attack trees suffer from a distinct lack of standards, prescriptive methodologies and common approaches in terms of their visual syntax. Attack trees are limited because they only represent a single attack, whereas an attack graph can represent multiple attacks. A full attack graph underlines all potential weaknesses and all possible attack paths in a certain network, while attack trees represent singular attacks, and attack forest, i.e. set of ATs is a way to solve this problem. [17]. Both attack graphs and attack trees are a graph-based representation of a cyber-attack

Another important difference between the two models regards their visual representation. First of all, in attack graphs the event flow is represented top-down, while in the majority of the attack trees this is depicted in a bottom-up way. [17]. Moreover, in the same sense, Attack Trees generally use vertices to represent exploits and not preconditions, while preconditions are assumed to have been met in the transition from one exploit to the next. Attack graphs represent both.

Essentially, both Attack Trees and Attack Graphs have a graph based structure. The main differences are: how the event flow is depicted, the representation of full and partial attacks and the representation of preconditions.

D. Network Security vs General case

During our research we identified two categories regarding the attacks represented by the Attack Graphs/Trees: the Network Defined attacks and the General case.

The first category is the Network Defined attacks. In this case, the works under study are focused on dealing with the Network Security, meaning that they engage only with network related attacks. We classify in this category only the papers that explicitly state that their work exclusively focus on representing Netphwork related attacks.

The second category denoted the General case, where the papers are also taking into account physical attacks or human

interactions. In many cases, the security of the system is compromised by human errors. Especially in large organizations where many people interact with extensive networks it is very common for the attackers to use social engineering or take advantage of human errors.

During our research we also discovered two papers [6], [7] referring to the term *Socio-technical* system. This term refers to a system involving: humans , machines and interaction with the environment [3]. The Socio-technical systems have five key characteristics as stated by Baxter et al. [3] and form their own category. In terms of attacks types covered by the Socio-technical systems, we found during our research that they represent also physical attacks, like social engineering and so we include them in the general case, but we state explicitly that the papers are referring to Socio-technical systems.

E. Categories

One motivation for this work was to contribute to the research community by identifying the tools and methods commonly used in order to automatically produce Attack Trees/Graphs. In view of this, we examined every paper and identified the main underling technology or method used. During this procedure we managed to identify 7 categories among the papers included in this survey. Following, we present a list of the categories:

- *Logical Formulas:* A paper falls into this category when the main components used to represent the system are logical formulas.
- *Templates:* The authors are generating the Attack Trees/Graphs using templates.
- *Library based:* The Attack Tree/Graph is generated given a library.
- *Model Checking:* A model checker is used to formally describe the system and check some properties.
- *Transformation Rules:* A representation of the system is already available and some transformation rules are applied in order to obtain the Attack Tree/Graph.
- *Artificial Intelligence:* Artificial intelligence techniques and algorithms are used in order to obtain the desired result.
- *Reachability:* The Attack Tree/Graph is based on the reachability of the nodes of the network.

We are going to examine 7 dimensions in each category:

- *Proofs:* If a paper includes formal proofs for the algorithms applied.
- *Code:* If a paper includes a reference to the implementation of the proposed solution and if the code is available online.
- *Prerequisites:* The prerequisites one should acquire in order to be able to apply the proposed solution.
- *Network Defined attacks, Sociotechnical or General:* If a paper proposed a Network Defined, Sociotechnical or General solution.
- *Graph or Tree:* If the proposed solution results to a Tree or a Graph.
- *Experiments:* If the authors conducted experiments.
- *Scalability:* If the authors present the scalability of their solution.

III. Research Method

In this section we are presenting the research method adopted in order to identify our final pool of papers we used in this survey. Following, we describe the research method, the study selection and the research questions.

A. Research Questions

Our work is aiming to examine and discuss the current literature for automatic generation of Attack Trees and Attack Graphs. Underlined below are the research questions that motivated us to conduct this survey.

- **RQ1:**Which techniques are currently used in the field?
- **RQ2:**What kind of attacks are taken into account (General or Network Defined) and what kind of evidence are presented to support the proposed solution (Experiments, code, mathematical proofs, scalability)?
- **RQ3:**Which are the challenges/limitations we identified in the field?

B. Research Method

We decided to conduct our research using the snowballing technique [32]. The snowballing technique refers to the procedure of identifying relevant papers from the reference list or the citations of a selected paper. Our final pool includes 21 papers. The steps of the procedure we followed in order to identify our final pool of papers are the following:

1) Start Set: The first step is forming the start set of papers. Firstly we had to identify relevant keywords to form a query to the selected database. The keywords we selected to use are: “Automatic Generation”, “Automated Generation”, “Attack Trees” and “Attack Graphs”. We performed our research in DTU Findit https://findit.dtu.dk, which is an open (guest access) database and includes publications from well known journals and databases. We used the following query: title:(“Automatic generation” OR “Automated generation”) AND title:(“Attack trees” OR “Attack graphs”). Our search return 13 papers, from which 3 where not available online. So based on relevance we formed our start set including five papers [1], [19], [26], [30], [33].

2) Iterations: After finding the start set, we have to decide which papers we are going to include in our final pool. For this purpose we applied *Backward and Forward Snowballing*. Backward snowballing refers to the examination of the reference list of the papers. In order to identify if a paper will be included we extract some information regarding the title, the author and the publication venue. Naturally, we should also take into account the context in which the paper is referenced. At this point, if a paper was still into consideration, we read the abstract and other parts of the paper in order to decide if the paper will be included. The forward snowballing is conducted in order to identify papers from the citation list of the paper being examined. Again, for each paper in the citation list, we

extracted some basic information, we took into consideration in which context the citation is taking place and for the final decision we read parts of the paper [32].

IV. Related Work

In this section we present other surveys focusing on the Automatic Generation of Attack Trees/Graphs. We also include a table to graphically represent some main characteristics of the paper included at the Related Work section. On Table I four different characteristics are presented. We denote the symbol ✓when the referred paper fulfills the corresponding characteristic. The first column includes the reference to the paper under examination, the second column "After 2020" refers to the papers that have been published after 2020, the third column "Automatic Generation" refers to the papers that take examine the Automatic Generation of Attack Trees/Graphs, the fourth column refers to whether the paper present classification for the papers included in their survey, based on the techniques used in each paper and the last one if the paper indicates the challenges identified in the field.

Kordy et al. [15] are presenting a survey focusing on DAG based graphical models for security. Their work summarize the state of the art of the existing methodologies, they compare them and classify them based on their formalism. Although this is a very extensive survey, they do not focus on the Automated generation of these models.

Lallie et al. [17] are examining the effectiveness of the visual representation of Attack Trees and Attack Graphs. They analyse how these structures are representing the cyber-attacks. They conclude that there is not a standard method to represent the Attack Trees/Graphs and that the research community should turn their attention on standardizing the representation. Although this paper is a great contribution, it does not examine the issue of the Automatic generation of these structures.

Wojciech et al. [31] are focusing on a survey regarding the application of formal methods on Attack trees, their semi-automated or automated generation and quantitative analysis of Attack Trees. Although they are referring to the Automatic generation, their research in not only focused in this part. Also they are not referring to the automatic Generation of Attack Graphs and their survey was conducted before 2020.

Taking everything into consideration, in our work we are trying to provide a wider overview for the Automatic Generation of Attack Trees/Graphs.

V. Overview - Quantitative Study

In this section we are giving an overview of the field through a quantitative study. The reader can find information regarding the quantity of the papers taking into account specific characteristics. The quantitative study allows us to have a better view of the field and the lack of specific information.

A. Publisher and Publication Year

In this part we provide statistics regarding the publishers and the publication year of the papers. Our goal is to identify the most popular venues and how the interest of the scientific community regarding the Automatic lower case, check in other places of the paper, please :) generation of Attack Trees and Attack Graphs altered during the years.

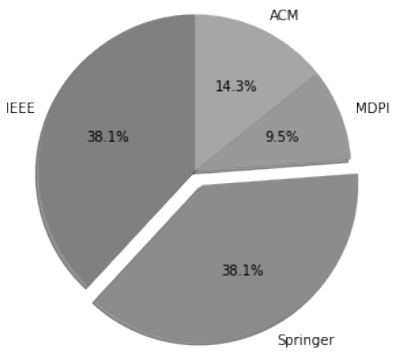

Fig. 3. Percentage of papers published by each publisher

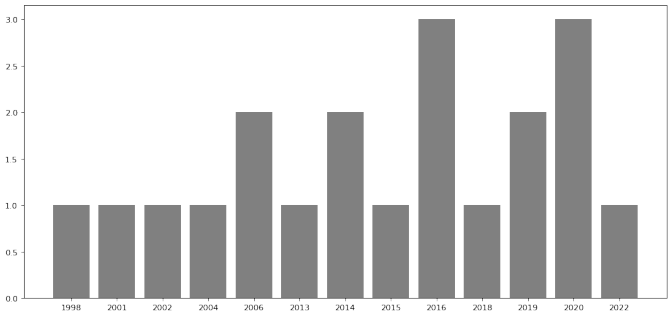

Fig. 4. Percentage of papers published each year

We can observe in Figure 3 that the papers included in this survey have been published by 4 different publishers - IEEE, Springer, ACM, MDPI. The publishers with the most publications on the filed is IEEE and Springer. Also in Figure 4 we can see the distribution of the papers through the years. The research community turned their interest to the field on 1998 and, at intervals, it has been active since. The interest has decreased between 2007 to 2012, but since then the research community seems to be more active on the specific field. For better visualisation the years with zero publications are not presented in the diagram.

B. Percentage of papers in each category

In this work we classify the papers into 7 different categories. Here we provide information about how many papers are in each category.

We can observe in Figure 5 that most of the papers fall into the Model checking category. Signifying that he research community preferred to use Model checking techniques for the Automatic Generation of Attack Trees/Graphs.

Later, on the paper we examine each category separately and provide the reader with proper information, in order to establish a better understanding on each one.

TABLE I
OVERVIEW OF THE PAPERS INCLUDED IN THE RELATED WORK

Paper	After 2020	Automatic Generation	Techniques' Classification	Challenges
[15]			✓	✓
[17]	✓			✓
[31]		✓		✓
Our Work	✓	✓	✓	✓

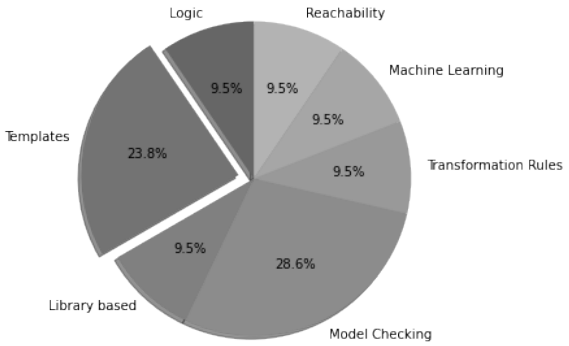

Fig. 5. Percentage of papers on each category

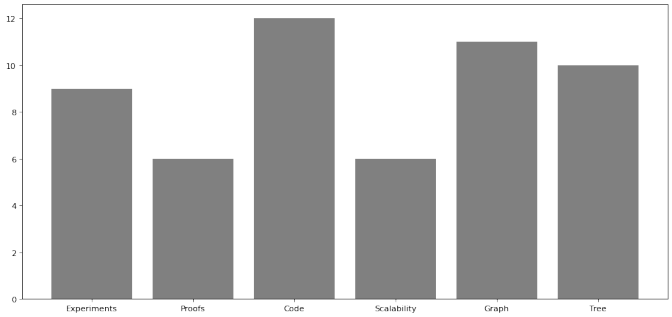

Fig. 6. Percentage of papers on each category

C. Experiments, Proofs, Code, Scalability

In this section we are going to present how many papers provide experiments, proofs, code or the scalability of their solution. In our perspective, these four characteristics are key information for the reader and a very supportive evidence for the quality of the proposed solution. We also present information about how many papers are referring to automatic generation of Attack Graphs and how many papers are referring to the Automatic Generation of Attack Trees. We concluded that almost half of the papers are presenting Attack Graphs and the other half Attack Trees. We can see the results on Figure 6.

We can observe on Figure 6 that less than half of the papers are presenting experiments or proofs and even less refers to the scalability, in terms of complexity of their solution.

VI. CLASSIFICATION BASED ON THE CATEGORIES

In this section, we present out findings for each category. At the beginning of each category we present a brief overview of our findings, following with a detailed presentation of each paper. We also include tables to graphically represent the key characteristics of every paper. On Tables II-VIII eleven different characteristics have been identified. We denote the symbol ✓when the referred paper fulfills the corresponding characteristic. The first column includes the reference to the paper under examination, the second column "Network Defined" refers to the papers only including network related attacks, the third column "General Attacks" refers to the papers that take into account multiple attacks at the automatic generation procedure, the fourth column refers to whether the paper presents Attack Trees and the fifth whether the paper presents Attack Graphs, the sixth whether the paper presents experiments, the seventh whether the code of the proposed solution is available, the eighth whether the paper presents proofs, the ninth if there is a reference to the scalability in terms of complexity, the tenth if the proposed solution requires any prerequisites and finally the eleventh indicates the year that the paper was published. The later is the only one filled with a number and not a ✓.

A. Logical Formulas

All of the papers falling in this category are using Logic as the basic technique to generate the Attack Tree/Graph.

It may be a bit of an overkill to present this detailed statistics when the category just consists of 2 papers. Let's discuss this. One way to go could be to provide a shorter summary, focusing on what really is worth remarking. In this category 9.52% of the papers are included. We can see that the range of publication years are from 2006 to 2014. All of the papers are referring to the scalability of their solution. We also can see that 1 out of 2 (50%) papers are generating Attack Trees and 1 out of 2 (50%) is generating an Attack Graph. Additionally, it is worth pointing out that 1 out of 2 (50%) papers are presenting solution only related to the network security and only 1 out of 2 (50%) is presenting a general approach regarding to attacks taken into account. Furthermore, 1 out of 2 (50%) papers have the code available online and 1 out of 2 (50%) papers present experiments. All of the papers are providing formal proofs for their solution. Finally, 1 out of 2 (50%) papers present solutions that require some prerequisites. Below we are presenting the papers included in this category.

Vigo et al. [30] propose a static-analysis approach. The authors use the Quality Calculus as specification language. The processes are translated to proposition formulae, demonstrating the connection between channel knowledge and program point accessibility. Then, the authors propose a backward

TABLE II
OVERVIEW OF THE PAPERS INCLUDED IN THE CATEGORY *LOGIC*

Paper	Network Defined	General Attacks	Attack Tree	Attack Graph	Experiments	Code	Proof	Scalability	Prerequisites	Year
[30]		✓	✓			✓	✓	✓		2014
[21]	✓			✓	✓		✓	✓	✓	2006

chaining search to the formulae from a program point l in order to generate the attack tree. The backward chaining exposes all the paths leading to point l, thus unveiling all the information needed to reach that point. The proposed way, defines the system in a formal manner and the threat scenarios are derived automatically by the definition of the system. Moreover, process calculi has been used broadly for defining software systems and organizations in a coherent way. It is worth pointing out that the authors are studying Attack Trees, but their approach can be also applied to produce Attack Graphs. They also refer to the scalability saying that the worst case is exponential, but this case does not occurs systematically as in the model checking approaches. The authors also share their code on $http://www.imm.dtu.dk/\sim rvig/quality-trees.html$. As a future work they state that they want to compare their approach with existing ones. The limitation of this approach is that they can only check if something has been received in the channel, but not the content. Finally, their is not a reference to the soundness of the result and the generated Attack Trees are mostly used for quantitative analysis.

Ou et al. [21] present a work based on MulVAL. MulVAL is a system for automatically identifying security vulnerabilities in networks. The central notion of MulVAL is that the configurations can be represented as Datalog tuples and security semantics and most attacks can be represented as Datalog rules. Every rule can be viewed as a logical formula. It evaluates the Datalog rules using XSB, a prolog system. The authors modified the MulVAL system to record the trace of the evaluations and send it to the graph builder. In every successful query a function notes the successful deviation to the trace file. The paper also introduces an algorithm to transform the traces into an attack graph. The transformation takes every trace step and transforms it to a part of a graph. This work is focused on the Network Security, meaning that the Generated Attack Graph do not include General Attacks. The proposed solution is supported with related formal proofs and experiments, but the code of the proposed solution is not available. Also, the generation of the Attack Graph is taking place in quadratic time. It is worth pointing out that the size of the generated logical graph is polynomial to the size of the network. In order to generate an Attack Graph using the proposed solution the are three prerequisites: Network configuration, machine configuration and security advisories.

B. Model Checking

All the papers in this category are using Model Checking to automatically generate the Attack Tree/Graph.

In this category 28.57% of the total papers. We can observe from the Table III that the chronological range of the publications is from 2002 to 2020. All of the papers have some prerequisites. Only 1 out of 6 (16.67%) papers has a reference to the scalability of the solution. Also, only 1 out of 6 (16.67%) papers present proofs and provides the code of the proposed solution. Furthermore, 4 out of 6 (66.67%) are presenting experiments, but none of them provides the code. This is a drawback since the reader cannot reproduce the experiments. Concerning the attacks included, 4 out of 6 (66.67%) papers are focused on the network security and the rest 2 out of 6 (33.34%) are presenting Attack Trees with general attacks. On this note the papers presenting General Attacks are only using Attack Trees as a graphical representation model and the rest of the papers present Attack Graphs and are focused on Network Defined attacks.

Sheyner et al. [27] in an earlier work presented a method to generate attack graphs automatically, with the use of model checking [26]. The authors choose six network components in order to construct network attack models: a set of host, a connectivity relation, a trust relation among the hosts, an intruder model, a set of individual actions the intruder can exploit to design an attack and an intrusion detection system. In order to construct the set of actions the intruder can exploit, the authors use real world vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database. In order to construct the attack graph, the toolkit checks a security property with model checking, in order to ensure that the property is satisfied. An overview of the toolkit is also presented, along with the main components and the user interface. The authors focus on Network Defined attacks. They do not present proofs, the scalability of the solution or the code. However, they present experiments. The required prerequisites are: the network topology, configuration data for each networked host and a library of attack rules. As a future work the authors want to specify a library of actions based on a vulnerability database provided to us by SEI/CERT. The current paper outlines a toolkit.

Sheyner et al. [26] their work present the network as a finite state machine. They have modified the model checker NuSVM in order to automatically generate the attack graph. First they form a security property. The model checker takes as input the property and the model M (the network). If the property is satisfied the model checker returns true, otherwise it provides a counterexample. The counterexample depicts a path that the attacker can follow to violate the security property. As a future work, they aim to generate attack graphs for a more general class of properties, apart from security properties. In this work the state of the model is depicted as a set of booleans representing the configuration of the network and

TABLE III
OVERVIEW OF THE PAPERS INCLUDED IN THE CATEGORY *MODEL CHECKING*

Paper	Network Defined	General Attacks	Attack Tree	Attack Graph	Experiments	Code	Proof	Scalability	Prerequisites	Year
[27]	✓			✓	✓				✓	2004
[26]	✓			✓	✓		✓		✓	2002
[23]		✓	✓			✓			✓	2016
[2]	✓			✓	✓			✓	✓	2020
[9]	✓			✓					✓	2019
[6]		✓	✓		✓				✓	2015

the attacker's actions as state transitions. As a consequence, the state-space created is exponential to the number of the system's variables. This work involves Attack Graphs focused on Network Defined attacks. The authors present proofs and experiments, but do not present the code and the scalability of their solution.

Pinchinat et al. [23] present ATSyRA. ATSyRA is a tool implemented on top of Eclipse to help the security experts interact with a user friendly environment when designing Attack Trees. The main motivation of the implementation of the tool was the security of military buildings. As a first step, the security expert has to define the system: the building description, the attacker's strength and their attack objective. The second step is to run the generation of the attack scenarios. The current step compiles the input specification into an attack graph. As a third step, the security expert is specifying a set of high level actions (HLA). The HLA imply how it can be refined into sub actions. The final step is to run the Attack Tree synthesis. This step uses the information given in step 3 and the graph produced in step 2, to construct the Attack Tree. The underlying interface for achieving this is a model checking algorithm. As a future work the authors want to examine other inputs from other fields. The paper presents General Attacks and create Attack Trees. The authors also provide the code in the form of a tool (https://atsyra2.irisa.fr/). The scalability, proofs and experiments are not provided in the paper. The description of the system is set as a prerequisite. They also state that a fully automated procedure may result in unsound results. We can observe that the solution is not fully automated since the security expert is also taking an active part in the procedure.

Ghazo et al. [2] introduce an algorithm for automatic graph construction and visualization that exploits an existing tool of model checking. In this work a tool for the description of architecture is taken into account. Its aim is creating the attack graph list of existing sequences in which atomic-level vulnerabilities could be used as a way to threaten the security of the system under consideration. The present tool can generate a representation of the system, i.e. network level, but also atomic vulnerabilities, post conditions and properties regarding security that can be valuable for our security purposes. Moreover, a model checker is used to identify in an automated way, the sequence of the attack as a counterexample. Then the counterexamples are parsed and iterated until attack sequences are found. Finally, a tool for visualization is used, to generate the graph. Essentially, this work introduces the development of a tool in the form of a graphical model to capture the vulnerabilities of an analyzed system by automatically constructing a graph exploiting features of an existing model checker.

Ibrahim et al. [9] present a solution including Hybrid Attack Graphs (HAG). Generally, HAGs illustrate the alteration of logical and real parameter values of an attacked system, as well as some recovery actions. Automated HAGs are generated automatically and can be visualized with a Java based tool. This procedure requires a formal description of the system's model and the security property (AADL language), validated by a model checker named JKind. This work offers a proof on how AHAGs can obtain logical changes in the parameters of the specific system under attack. In essence, the authors present a binary classification problem and a multi-output learning algorithm in order to generate an attack graph starting from some given information on an analyzed system and a network.

David et al. [6] present a methodology for the modeling of socio-technical attacks and systems, namely those that are related to human behaviour. With the use of timed automata and automated model checking, it is possible to obtain in an automatic manner the possible attacks regarding the considered model. You may also do simulations and analyze the obtained attacks or the model itself. Finally, state-of-the-art tools can be exploited to analyze models by employing timed automata in this methodology. In this methodology, model checking is included so that we can spot attacks against the analyzed system. In the end, a complex example is provided, describing an application of this procedure.

C. Templates

In this category belong the papers using templates to generate the Attack Trees/Graphs.

This category includes 23.80% of the total papers. We can see that the year of publication vary from 1998 to 2020. We can observe on Table IV that all of the papers require prerequisites. Also, 3 out of 5 (60%) papers are limited to Network Defined attacks, the rest of them (40%) are dealing with General attacks. Furthermore, 2 out of 5 papers are constructing an Attack Tree, while 3 out of 5 are constructing an Attack Graph. Only 1 out of 5 (20%) of the papers is presenting experiments, the same percentage includes proofs. Additionally, 1 out of 5 (20%) papers are sharing the code of their solution. It is important to point out that none of

TABLE IV
OVERVIEW OF THE PAPERS INCLUDED IN THE CATEGORY *TEMPLATES*

Paper	Network Defined	General Attacks	Attack Tree	Attack Graph	Experiments	Code	Proof	Scalability	Prerequisites	Year
[5]	✓		✓		✓	✓			✓	2020
[28]	✓			✓					✓	2002
[29]		✓		✓			✓		✓	2014
[16]	✓		✓						✓	2020
[22]		✓		✓					✓	1998

TABLE V
OVERVIEW OF THE PAPERS INCLUDED IN THE CATEGORY *LIBRARY-BASED*

Paper	Network Defined	General Attacks	Attack Tree	Attack Graph	Experiments	Code	Proof	Scalability	Prerequisites	Year
[24]		✓	✓			✓	✓	✓	✓	2020
[13]		✓	✓			✓	✓		✓	2018

the papers are referring to the scalability of their proposed solution.

Bryans et al. [5] introduce a method to generate attack trees automatically from the description of the network and a set of templates. Each template represents an attack step. The templates are using variables that have to be replaced by the components of the system under investigation. The proposed algorithm is recursive, at each step a leaf of the template trees is investigated and expanded if the leaf contains an unbounded variable. If the name of the child matches the name of the root of one of the templates, it is replaced. When, the unbounded variable matches multiple templates an OR node is introduced. The authors are constructing an Attack Tree and the code is available online at https://tinyurl.com/uoptgfb. They also present the experiments conducted, but they do not refer to the scalability and the proofs. As a future work they aim to expand the method to support more networks, since right now they are focused on automotive communication networks.

Swiler et al. [28] proposed a method to generate an attack graph, in which each node represents a state of the network. The tool has as an input the configuration of the network, the attacker's profile and the attack templates. The attack templates represent steps of known attacks or strategies of moving from one state to another. The tool combines the input information and customizes the generic template attacks according to the attacker profile and the network configuration. Every graph includes some variables that represent the state of the network. When a node in a graph matches the requirements of the template a new edge is added to the network and new nodes are created. This paper is dealing with Network Defined attacks. The authors are not providing experiments, proofs, code or refer to the scalability of their solution.

Tippenhauer et al. [29] are presenting a Goal System Attacker graph. There is a 3-step procedure to generate it. At first the security goal and the workflow of the system are used to produce the G-graph. The result is used in combination with the system description to generate the GS-graph, which is then combined with the attacker model to generate the GSA graph. The authors used the framework described above and noticed that there is a series of patterns. These patterns were used to implement some templates. Afterwards, they defined local extensions to progressively generate these graphs using the predefined templates. If there is a matching node, the template is being integrated to the main graph. The authors are taking into account General attacks and they present proofs for their work. They do not give an input for the scalability, the code or any experiments conducted.

Kumar et al. [16] introduce the use of an AT template as a feature diagram, i.e. formal and graphical notations, to solve the non-standardization problem of Attack Trees. This problem states that, in general, a standard template for Attack Trees design does not exist. The template considered in this work is structured in layers each refining the previous ones, in order to construct tree semi-automatically. This Attack Tree template can be seen as an abstraction that can capture crucial scenarios about attacks by refining hierarchical relationship rules. Moreover, the template is constructed by going through the literature on Attack Trees so that common characteristics in their design can be found. Summing up, the authors present a way for identifying proper meta-categories suitable for Attack Trees by means of feature diagrams, in order to construct the trees in a semi-automated manner.

Phillips et al. [22] present a system where the graph is generated using the attack profile, the input templates and the configuration of the system. The procedure starts from the goal node and is built backwards. Then, we search the templates to find an edge that the head matches the goal node. The paths that do not satisfy the attacker profile are eliminated. The procedure is being repeated until we reach an initial state that is not the head of an edge. The authors claim that their approach can model dynamic aspects by overwriting the configuration of the network. As a prerequisite, besides the configuration of the network and the attacker's profile, a database with common attacks is also required. They do not present the code, proofs, experiments or a reference to the scalability.

D. Library Based

All the papers included to this category are constructing the Attack Trees/Graphs given a Library.

TABLE VI
OVERVIEW OF THE PAPERS INCLUDED IN THE CATEGORY *Artificial Intelligence*

Paper	Network Defined	General Attacks	Attack Tree	Attack Graph	Experiments	Code	Proof	Scalability	Prerequisites	Year
[14]	✓			✓	✓				✓	2022
[4]	✓			✓						2019

TABLE VII
OVERVIEW OF THE PAPERS INCLUDED IN THE CATEGORY *REACHABILITY*

Paper	Network Defined	General Attacks	Attack Tree	Attack Graph	Experiments	Code	Proof	Scalability	Prerequisites	Year
[10]	✓			✓	✓			✓	✓	2006
[8]	✓		✓		✓			✓	✓	2013

In this category of the papers are included 9.52% . We can see that the year of publication range from 2018 to 2020. All of the papers are taking into account General Attacks and are constructing Attack Trees. Also, all of the papers are presenting the code and proofs supporting their solution and require some prerequisites. Finally 1 out of 2 (50%) of the papers are referring ti the scalability.

Pinchinat et al. [24] are constructing an Attack Tree. The main goal of this work is to construct the attack tree that explains a trace given a library L (a set of refinement rules). This approach is also good for forensics, but one should have the logs. The attack tree is being build according to the refinements provided by the library. The algorithm that handles the procedure is based on the CYK algorithm, which answers whether some input context free grammar can generate some input world. The code is also available for the readers on http://attacktreesynthesis.irisa.fr/. The procedure is Semi-automatic to avoid unsound results. Also the Attack Tree consists of AND, OR and SAND refinements. The authors are also presenting proofs for their solution. The mentioned scalability is that the algorithm is polynomial at the size of the trace. The attacks taken into account are General. Finally, the authors are not presenting experiments.

Jhawar et al. [13] presented a work focused on an semi-automatic procedure for creating attack trees. The construction of the Attack trees can be summed up in four steps: First step: A group of experts define an initial version of the tree. Second step: Some automatic mechanisms are used to enhance the tree. Third step: The experts curate the new version of the tree. Fourth step: Repeat step two and three. The paper is focusing on implementing the second step. The authors express a predicate based annotation of the tree, that to determine if an attack tree can be attached to another attack tree as a sub-tree. The authors construct a library of annotated attack trees using the National Vulnerability Database. After that, they use this library to extend attack trees manually constructed, using the Common Attack Pattern Enumeration and Classification. As a future work, they want to implement a dynamic library and expand the idea in other fields like counterexamples. The code is available online at https://github.com/yramirezc/lib-annotated-attack-trees. The authors are also providing the reader with proofs for their solution. Finally, they do not present experiments or the Scalability of their solution.

E. Artificial Intelligence

All the papers assigned in this category are using Artificial Intelligence in order to construct the corresponding Attack Trees/Graphs.

In this category are included 9.52% of the papers included in this survey. The range of the publication years varies from 2019 to 2022. We can observe that all the papers are dealing with Network Defined attacks and are constructing the corresponding Attack Graph. Also none of the papers share the code of their solution. Furthermore, 1 out of 2 (50%) papers are presenting experiments and have prerequisites. None of the papers in this category are presenting proofs or the scalability of their approach.

Koo et al. [14] introduces a method to support the generation of an attack path given a set of attack graphs by using Deep Learning and Machine Learning. The attack graph is obtained by training the model. It is important to note that, in order to achieve our goal, the topology of the network and system information are needed. We then apply feature extraction to acquire an attack graph generation model by exploiting the input data. Finally, the authors are using an evaluation metric to evaluate the predicted path. To sum up, the authors present a binary classification problem together with a multi-output learning algorithm, in order to generate an attack graph starting from information of the system and the network.

Bezawanda et al. [4] presented a tool that automatically generates the PDDL representation of an Attack Graph from descriptions found in the CVE or the NVD system. The authors are using natural language processing to produce the PDDL from the textual description. Also the tool offers the transformation of the PDDL to the corresponding Attack Graph for visualization purposes. The procedure can be explained in 4 steps. First step: Extract information from the Vulnerability database and form the PDDL domain. Second step: Form the PDDL problems using the PDDL domain and event logs of the system. Third step: An Artificial Intelligence algorithm is generating a PDDL plan for each PDDL problem. Fourth step: The tool updates the content of the PDDL domain in every modification of the input data. It is also worthy to point out what is a PDDL domain and a PDDL problem. The

TABLE VIII
OVERVIEW OF THE PAPERS INCLUDED IN THE CATEGORY *TRANSFORMATION RULES*

Paper	Network Defined	General Attacks	Attack Tree	Attack Graph	Experiments	Code	Proof	Scalability	Prerequisites	Year
[7]		✓	✓						✓	2016
[11]		✓	✓						✓	2016

PPDL domain is some problem descriptions with the corresponding actions and constraints. The PPDL domain includes abstract variables. When these variables take a specific value, an instance of the PDDL domain is created which is called PDDL problem. The PDDL problem is solved with the help of the PDDL planner, that is trying to find a plan to satisfy the PDDL problem (sequnce of action one must perform to achive the end goal). The authors do not present experiments, code, proofs or the scalability of their solution. Finally, we should point out that there are no required prerequisites.

F. Reachability

All the papers included in this category are using the Reachability of the nodes in the network in order to construct the corresponding Attack Tree/Graph.

This category includes 9.52% of the papers. The years of publication ranges from 2006 to 2013. All of the papers in this category are presenting experiments and show the scalability. Also, all of the papers require some prerequisites and deal with Network defined attacks. Furthermore, 1 out of 2 (50%) papers is constructing an Attack Tree and the other one an Attack Graph. None of the papers are providing the reader with their code or formal proofs of their solution.

Ingols et al. [10] proposed a system based on multiple prerequisite graphs. They argue that this type of graphs are better that the full graphs or the predictive graphs due to the lack of dependencies. The system processes the input data (Map of the network) and computes the reachability matrix. The computation of the reachability matrix is equipped with some improvements crucial to saving memory and time. Some sections of the matrix are collapsed into reachability groups. The filtering rules are replaced by Binary decision diagrams, resulting to filtering rules in linear time. The graph is constructed using a breadth first technique. Multiple prerequisite graphs consist of three different kinds of nodes; state nodes, prerequisite nodes and vulnerability instance nodes. During the construction, every type of node is being added differently to the graph. Also, this work presents a graph simplification for visual presentation. The proposed solution import data from: Nessus, Sidewinder and Checkpoint firewalls, the CVE dictionary and NVD. The generation of the Attack Graph depends on data that can be obtained quickly. The data are evaluated and reported as early sa possible. This work assumes that the paths are monotonic, meaning that the attacker will never go back. Also, the authors are not modelling Client side attacks. The authors refer to the scalability in terms of complexity as almost linear to the size of the network. Also, they present experiments for their solution. Finally, it is also worth mentioning that they do not present the code or proofs for their work.

Hong et al. [8] identify all the paths in the network of the system to construct the full AT, which consists of AND and OR gates. After the construction of the full AT there are two proposed methods: 1. Simplified Attack Tree with Full Path Calculations 2. SATwIPC simplified Attack Tree with Increental Path calculation. The first method requires a logical expression of the attack tree and removes the sequence information from the expression but it groups similar nodes. The second method is maintaining attack path information and is constructed by exploiting network configurations and sequences of vulnerabilities. The authors present Attack Trees that depict attacks concerning only the Network Security. The authors do not present formal proofs or the code of their solution, but they present the scalability and experiments. Finally, the proposed solution has one prerequisite the description of the network.

G. Transformation Rules

In this category the papers that are using Transformation Rules to obtain the corresponding Attack Tree are included.

This category include 9.52% of the papers included in this survey. Also, the publication year for all the papers is 2016. All of the papers included in this category are taking into account General Attacks and construct the corresponding Attack Tree. Furthermore none of the papers in this category are presenting experiments, the code, proofs or scalability of the proposed solution.

Gadyatskaya et al. [7] presents an Attack-Defense tree generation for socio-technical models. These trees represent both the attacker's options and the available countermeasures. This work presents the creation of a group of Attack-Defense bundles given a so io-technical model. These bundles can be used for generating Attack-Defense Trees. In this context, socio-technical models are important because they can capture General attacks, also in large organizations. Basically, the aim of this work is the automation of Attack Tree construction, exploiting socio-technical models.

Ivanova et al. [11] are aiming to transform a graphical system model to a graphical attack model. The graphical system model includes: locations, actors, processes, items. The actors and the processes can be decorated with policies and credentials. For every component the authors introduce some guidelines for transforming the graphical representation to the corresponding Attack Tree. As a future work the authors would like to extend their model to include attacks aiming to disturb the environment of the system, where current solution deal with confidentiality and integrity.

VII. Challenges and Future Directions

After classifying and examining different characteristics of every paper we identified some challenges in the field. In this section we present the challenges identified that can serve as future direction for the research community.

- *Use of SAND:* In the automatic generation of Attack Trees only a few papers are including the $SAND$ operator. The $SAND$ operator can represent multiple situations that occur in different kind of attacks [12]. So we believe that it is important to also include this operator at the automatic generation.
- *Use of XOR:* There is no reference is the literature of the exclusive choice operator. There are cases in some attacks that an attacker will end up with an exclusive choice. For example to break one out of two doors, due to the equipment available - i.e. one bomb available. In our opinion it is necessary for the research community to include such an operator on the Attack Trees. With the addition of the XOR operator the security experts or the programs analysing the potential attacks can exclude some paths or traces.
- *Dynamic Solutions:* The systems are being upgraded constantly. New variables or configurations can make an Attack Tree/Graph useless. The need for more dynamic solutions are crucial. There are a few papers in the literature dealing with this issue.
- *Poor Semantics / Proofs:* The use of semantics and proofs is really important. There are some works that do not use proofs or semantics to support their solution. Neglecting the semantics can lead to poor or unsound results. The use of proofs can help the reader better understand the concepts used and help the researchers expand the field. Semantics can help the community develop the same language concerning the generation of Attack Trees/Graphs.
- *Forensics:* Another field important to the security experts are forensics. Forensics can help the security experts identify vulnerabilities that did not previously taken into account and secure better their systems. Using forensics one can generate Attack Trees/Graphs to depict what when wrong in a system.
- *Sound Results:* The automatic generation of Attack Trees/Graphs provide a very good tool for the security experts. But it might produce unsound results that are not usable. There are not many works providing proper means to prevent the generation of unsound results. Mostly semi-automated procedures have been proposed to avoid the generation of unsound results.
- *Prerequisites:* Most of the papers require some prerequisites in order to generate the Attack Tree/Graph. This might be tedious for the security experts (since they might follow a semi-automated procedure, that requires their involvement), but a fully automated procedure might produce unsound results. Some prerequisites can be the configuration of the system, which is a very important file that an attacker can exploit to take control of the system. So the prerequisites can themselves be a vulnerability. It is important to find the right balance between the amount of the prerequisites and the sound results. It is also important to examine the nature of the prerequisite and how crucial they can be for the security of the system.
- *Scalability:* The Scalability is one of the main characteristics of a tool. Nowadays we use very large networks to cover our needs. The automatic generation of Attack Trees should be adapted to be able to perform under these circumstances. So, it is important to study the scalability and try to optimize the proposed algorithms.
- *Attack Defence Trees:* A few papers are investigating the Automatic generation of Attack Defence Trees. After the automatic generation of Attack Trees/Graphs, is it natural to investigate the generation f the corresponding defences/countermeasures.

VIII. Conclusion

Graphical Representation Models are widely used in the security field to depict all possible attacks of a system. Two of the most common Graphical representation models are Attack Trees and Attack Graphs. These structures provide a user friendly representation for the security experts and in parallel they constitute a useful tool for analyzing the systems under investigation. Designing these kind of graphical representations by hand can be error-prone and tedious for the security experts. Consequently, the research community turned their interest in finding ways to automatically generate these structures. In this work we present the state of the art of the automatic generation of Attack Trees/Graphs. We structured our survey to answer 3 research questions: **RQ1**, Which techniques are currently used in the field, **RQ2**, what kind of attacks are taken into account and the evidence the authors present to support their solution (experiment, scalability, code, mathematical proofs), **RQ3**, what kind of limitations we identified in the field. We answered **RQ1**, by classifying the papers into 7 different categories in Section VI and presenting the categories with the percentage of their population in Section V. Following that, we answered **RQ2** in Section V, giving an overview of how many papers included the corresponding parameters and in Section VI we also present tables that show in detail which paper include which parameters. Finally, taken everything into consideration we answer **RQ3** in Section VII, where we present the limitations/challenges we identified in the field that also constitute future directions for the research community.

Acknowledgment

This work has been supported by Innovation Fund Denmark and the Digital Research Centre Denmark, through bridge project "SIOT – Secure Internet of Things – Risk analysis in design and operation".

References

[1] M. Ugur Aksu, Kemal Bicakci, M. Hadi Dilek, A. Murat Ozbayoglu, and E. ıslam Tatli. Automated generation of attack graphs using nvd. In *Proceedings of the Eighth ACM Conference on Data and Application*

Security and Privacy, CODASPY '18, page 135–142, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Alaa T. Al Ghazo, Mariam Ibrahim, Hao Ren, and Ratnesh Kumar. A2g2v: Automatic attack graph generation and visualization and its applications to computer and scada networks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 50(10):3488–3498, 2020.

[3] Gordon Baxter and Ian Sommerville. Socio-technical systems: From design methods to systems engineering. *Interacting with Computers*, 23(1):4–17, 2011.

[4] Bruhadeshwar Bezawada, Indrajit Ray, and Kushagra Tiwary. Agbuilder: an ai tool for automated attack graph building, analysis, and refinement. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 23–42. Springer, 2019.

[5] Jeremy Bryans, Lin Shen Liew, Hoang Nga Nguyen, Giedre Sabaliauskaite, Siraj Shaikh, and Fengjun Zhou. A template-based method for the generation of attack trees. In Maryline Laurent and Thanassis Giannetsos, editors, *Information Security Theory and Practice*, pages 155–165, Cham, 2020. Springer International Publishing.

[6] Nicolas David, Alexandre David, Rene Rydhof Hansen, Kim G. Larsen, Axel Legay, Mads Chr. Olesen, and Christian W. Probst. Modelling social-technical attacks with timed automata. In *Proceedings of the 7th ACM CCS International Workshop on Managing Insider Security Threats*, MIST '15, page 21–28, New York, NY, USA, 2015. Association for Computing Machinery.

[7] Olga Gadyatskaya. How to generate security cameras: Towards defence generation for socio-technical systems. In Sjouke Mauw, Barbara Kordy, and Sushil Jajodia, editors, *Graphical Models for Security*, pages 50–65, Cham, 2016. Springer International Publishing.

[8] Jin Bum Hong, Dong Seong Kim, and Tadao Takaoka. Scalable attack representation model using logic reduction techniques. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 404–411, 2013.

[9] Mariam Ibrahim and Ahmad Alsheikh. Automatic hybrid attack graph (ahag) generation for complex engineering systems. *Processes*, 7(11):787, 2019.

[10] Kyle Ingols, Richard Lippmann, and Keith Piwowarski. Practical attack graph generation for network defense. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 121–130, 2006.

[11] Marieta Georgieva Ivanova, Christian W. Probst, René Rydhof Hansen, and Florian Kammüller. Transforming graphical system models to graphical attack models. In Sjouke Mauw, Barbara Kordy, and Sushil Jajodia, editors, *Graphical Models for Security*, pages 82–96, Cham, 2016. Springer International Publishing.

[12] Ravi Jhawar, Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Rolando Trujillo-Rasua. Attack trees with sequential conjunction. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, pages 339–353, Cham, 2015. Springer International Publishing.

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

[14] Kijong Koo, Daesung Moon, Jun-Ho Huh, Se-Hoon Jung, and Hansung Lee. Attack graph generation with machine learning for network security. *Electronics*, 11(9):1332, 2022.

[15] Barbara Kordy, Ludovic Piètre-Cambacédès, and Patrick Schweitzer. Dag-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer Science Review*, 13-14:1–38, 2014.

[16] Rajesh Kumar. An attack tree template based on feature diagram hierarchy. In *2020 IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys)*, pages 92–97, 2020.

[17] Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. A review of attack graph and attack tree visual syntax in cyber security. *Computer Science Review*, 35:100219, 2020.

[18] Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In Dong Ho Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005*, pages 186–198, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[19] K. Muthumanickam and E. Ilavarasan. Automatic generation of p2p botnet network attack graph. In Vinu V. Das, editor, *Proceedings of the Third International Conference on Trends in Information, Telecommunication and Computing*, pages 367–373, New York, NY, 2013. Springer New York.

[20] Vidhyashree Nagaraju, Lance Fiondella, and Thierry Wandji. A survey of fault and attack tree modeling and analysis for cyber risk management. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–6, 2017.

[21] Xinming Ou, Wayne F Boyer, and Miles A McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345, 2006.

[22] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, page 71–79, New York, NY, USA, 1998. Association for Computing Machinery.

[23] Sophie Pinchinat, Mathieu Acher, and Didier Vojtisek. Atsyra: An integrated environment for synthesizing attack trees. In Sjouke Mauw, Barbara Kordy, and Sushil Jajodia, editors, *Graphical Models for Security*, pages 97–101, Cham, 2016. Springer International Publishing.

[24] Sophie Pinchinat, François Schwarzentruber, and Sébastien Lê Cong. Library-based attack tree synthesis. In Harley Eades III and Olga Gadyatskaya, editors, *Graphical Models for Security*, pages 24–44, Cham, 2020. Springer International Publishing.

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

[26] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 273–284, 2002.

[27] Oleg Sheyner and Jeannette Wing. Tools for generating and analyzing attack graphs. In *International symposium on formal methods for components and objects*, pages 344–371. Springer, 2003.

[28] L.P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 2, pages 307–321 vol.2, 2001.

[29] Nils Ole Tippenhauer, William G. Temple, An Hoa Vu, Binbin Chen, David M. Nicol, Zbigniew Kalbarczyk, and William H. Sanders. Automatic generation of security argument graphs. In *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pages 33–42, 2014.

[30] Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson. Automated generation of attack trees. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 337–350, 2014.

[31] Wojciech Wideł, Maxime Audinot, Barbara Fila, and Sophie Pinchinat. Beyond 2014: Formal methods for attack tree–based security modeling. *ACM Comput. Surv.*, 52(4), aug 2019.

[32] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, New York, NY, USA, 2014. Association for Computing Machinery.

[33] Shangqin Zhong, Danfeng Yan, and Chen Liu. Automatic generation of host-based network attack graph. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 1, pages 93–98, 2009.