Learning Attack Trees by Genetic Algorithms

Florian Dorfhuber[1,2(✉)], Julia Eisentraut[1], and Jan Křetínský[1,2]

[1] Technical University of Munich, Munich, Germany
florian.dorfhuber@in.tum.de, julia.eisentraut@posteo.de, xkretins@fi.muni.cz
[2] Masaryk University Brno, Brno, Czech Republic

Abstract. Attack trees are a graphical formalism for security assessment. They are particularly valued for their explainability and high accessibility without security or formal methods expertise. They can be used, for instance, to quantify the global insecurity of a system arising from the unreliability of its parts, graphically explain security bottlenecks, or identify additional vulnerabilities through their systematic decomposition. However, in most cases, the main hindrance in the practical deployment is the need for a domain expert to construct the tree manually or using further models. This paper demonstrates how to learn attack trees from logs, i.e., sets of traces, typically stored abundantly in many application domains. To this end, we design a genetic algorithm and apply it to classes of trees with different expressive power. Our experiments on real data show that comparably simple yet highly accurate trees can be learned efficiently, even from small data sets.

1 Introduction

The security of real-world applications depends on various factors. Both physical aspects and the IT infrastructure play a crucial role. Additionally, all humans who interact with the system need to use it securely. Consequently, rigorous threat modeling should provide insights and formal arguments to system designers and security *experts* as well as allow for easy communication of the findings to *everyday users* and all stakeholders without computer-science background. *Attack trees* [36] and their extensions [14,18,24,25] are valued for combining both aspects and, as a result, have seen an increasing number of applications recently [9,22,34,35]. In particular, to identify vulnerabilities earlier, modeling methodologies [38] often include attack trees as recommended, e.g., by *OWASP CISO AppSec Guide*[1], or by NATO's *Improving Common Security Risk Analysis* report [29].

Automated generation of attack trees has been recognized as the **major gap** between the needs of practitioners and the current state-of-the-art tools [10] since manual construction is tedious and error-prone. In recent years, some approaches to overcome this issue have been provided. However, none of them

[1] https://www.owasp.org/index.php/CISO_AppSec_Guide:_Criteria_for_Managing_Application_Security_Risks.

E. Ábrahám et al. (Eds.): ICTAC 2023, LNCS 14446, pp. 55–73, 2023.
https://doi.org/10.1007/978-3-031-47963-2_5

are automated, in particular, due to relying on (i) another system model to be given [5,10,17,31,32,39] or (ii) a library of models or refinement rules [5,19,31–33], which defers the workload rather than reduces it. All these approaches have in common that they construct an attack tree representing exactly the attacks given on input, not inferring other likely attacks but relying on the completeness of their description.

In contrast, in this work, for the first time, we *learn* an attack tree from a set of *sequential descriptions (traces)*, regarding the attack tree as a classifier of traces into successful and unsuccessful attacks. This has two advantages. Firstly, traces can be obtained not only by running other existing system models or by expert knowledge but also simply from *logs* of system behavior and attacks, which are often abundantly collected in many application domains. Moreover, these heterogeneous sources can also be combined, another valuable consequence of our weak assumption. Secondly, we only require these traces to be labeled as successful or unsuccessful attacks, but they *do not need to be consistent or complete*. In real-world systems, this liberty is an essential property since (hopefully) not every possible attack on the system has already been recorded, third-party software might not allow inspecting all necessary details or experts might disagree on possible attacks.

As the basis for our learning approach, we use *genetic algorithms*, randomized optimization algorithms inspired by the process of natural selection. Our algorithm maintains a set of attack tree models and optimizes them over a fixed number of *generations*. In each generation, models are altered (*mutation*) or combined (*crossover*) into new models, using simple editing operations on trees. Both our mutations and our crossovers only modify the structure of the attack tree (e.g., exchanging subtrees containing the same basic events or switching the position of basic events) but do not need any domain knowledge. Only the fittest models among the new ones (w.r.t. a *fitness function*) are transferred to the next generation. As the fitness function, we choose a weighted sum of the *sensitivity* and the *specificity* of the prediction quality of our attack tree models.

To evaluate our approach, we use both synthetic and real data. To measure the performance, we create sets of labeled traces from the attack trees and learn trees from those traces. We provide experimental evidence of the following claims, arguing for the usability of this first automatic approach, and observations giving insights into the nature of the problem and the data:

- Sanity check: Our approach can learn models perfectly predicting attacks if all successful attacks are given.
- Checking for overfitting: We run k-fold cross-validation to show that overfitting is a minor problem for the algorithm.
- For comparison (given that there are no automatic competitors), we also implement *logistic regression* to solve the task because it is known to perform well on binary classification tasks; however, logistic regression does not provide any graphical model, which is our main aim. Regarding *accuracy*, our genetic algorithms for attack trees are on par with logistic regression, and, in contrast to the latter, our algorithm provides a ***simple graphical model***.

- The learned trees are essentially as simple as the original trees used to generate the traces. In particular, for attack trees, the learned ones are of the same size. Note that this small size is crucial to ensure the *explainability* of the automatically constructed model.
- We show that the number of generations is the most relevant hyper-parameter regarding fitness.
- Interestingly, for our algorithm to learn accurate models, even a few traces of successful attacks are sufficient.

In summary, the approach is the first to generate attack trees automatically. The trees are very accurate, given our inputs are only sampled data. Moreover, surprisingly and fortunately, it comes at **no extra cost for the graphicality** compared to classification by logistic regression, where no graphical model is produced. (An explanation of why decision trees are not suited for the task is discussed in Subsect. 2.2).

Our contribution can be summarized as follows:

- We provide a genetic algorithm to learn attack trees and their extensions from labeled sets of traces, the **first to automatically generate these graphical models**. For comparison, we also implement logistic regression.
- In a series of detailed experiments, we show the feasibility of our approach, also on real benchmarks from the military domain.
- We provide an implementation and visualization of all algorithms in a tool that links to standard verification tools for attack trees. The output models can be directly analyzed using tools such as ADTool [11] or displayed using Graphviz[2]. This unleashes the power of the verification tools to attack tree users, eliminating the manual step of model construction.

Organization. We present related work in Subsect. 1.1. In Sect. 2, we recall the most important features of genetic algorithms and attack trees. We show how to define mutations and crossovers to learn attack trees in Sect. 3 and evaluate our algorithms in Sect. 4. Furthermore, we provide a real-world demonstration on the KDD Cup 1999 dataset in Sect. 4.7. In Sect. 5, we conclude the paper and present several ideas for future work.

1.1 Related Work

This section briefly presents the most recent literature on attack tree synthesis and the use of genetic algorithms in attack model generation and security in general. A recent overview on the *analysis of attack trees using formal methods* can be found in [40]. We refer the interested reader to [23] for a *broad overview on different extensions of attack trees*. A recent survey on using *genetic algorithm for multi-label classification* can be found in [12].

[2] https://graphviz.org/.

Genetic Algorithms in Attack Model Generation and Security. Genetic algorithms have seen some applications in security. To the best of our knowledge, there is no approach to attack tree synthesis based on genetic algorithms. In this section, we present the applications which are closest to our application. In [26], the authors add an attacker profile (which can be seen as a set of constraints) to the attack tree analysis. An attack is only successful in its setting if it satisfies all constraints of the attacker profile. The authors use a genetic algorithm to simplify the search for attacks satisfying the attacker profile, but a system designer constructs the attack tree itself. Similarly, genetic algorithms have been used in [13] to find a strategy to reduce the system vulnerability with as little cost as possible. Also, [21] uses genetic algorithms to approximate the maximum outcome for the attacker on a given attack tree. On attack graphs [37], genetic algorithms have been used to find minimum cut sets [2]. Another more prominent application of genetic algorithms in security is the generation of intrusion detection systems [28,30]. While these approaches often also take a set of sequential behavior descriptions as inputs, learning an understandable model plays a very minor role.

Linard et al. [27] propose a genetic algorithm to generate fault trees. It also uses data generated by a system with labels for system failure. This approach performed well in a benchmark with up to 24 nodes. Their data sets use between 9000 and 230k data points. Recently, [20] expanded the idea to a multi-objective approach. Compared to these publications, our algorithm needs less data with a few successful attacks to perform the task. Also, the KDD example in Sect. 4.7 includes more than twice the number of basic events in the largest model used in their experiments or a case study [6].

Generation of Attack Trees. We present the last decade of research on attack tree synthesis.

In [15], Hong et al. present an approach to automatically construct attack trees from a set of sequential descriptions of attacks. In the first step, an exhaustive model as disjunction over all given traces is built. Consecutively, the model is simplified by collapsing similar nodes without changing the logical meaning. In contrast to [15], we use both successful and unsuccessful attacks as inputs, which prevents our approach from overfitting to a certain extent. Hence, their approach needs all successful attacks to build a complete model, while our algorithm is likely to predict future attacks not explicitly given. In [39], Vigo et al. provide a generation method to synthesize attack trees from processes specified in value-passing quality calculus (a derivation of π-calculus). The authors define an attack as a set of channels an attacker needs to know to reach a certain location in the system. The attack specifications and the processes are then translated to propositional logic formulae interpreted as attack trees.

In [31,32], Pinchinat et al. present an approach to sequential attack tree synthesis from a library, a formal description of the system and an attack graph. While the attack graph is automatically constructed from model checking, the formal description consisting of a set of actions assigned to different abstraction levels and a set of refinement rules of higher-level actions into combinations of lower-level actions need to be specified by the system designer. Although reusing

libraries is possible, it is necessary to specify them manually at some point. This approach does not support the operator `AND`.

The synthesis approach presented in [17] by Ivanova et al. also transforms a system model into an attack tree. The method requires a graph-based model annotated with policies as input. We could not find a publicly available implementation. In [10], Gadyatskaya et al. present an approach to sequential attack tree synthesis that requires a specification of the expected behavior (called semantics) and a refinement specification as input[3]. Again, this approach does not feature the operator `AND`. Both the semantics (in the form of series-parallel graphs) and the refinement specifications are not automatically constructed in [10].

The approach presented in [19] by Jhawar et al. requires an expert to interact with the overall construction process of an attack tree to such an extent that it cannot be considered an automated generation method in comparison to the other approaches presented in this section. An attack tree, at least rudimentarily describing attacks on a given system and annotations to the nodes in the form of preconditions and postconditions need to be given to the approach. Using the annotations, the attack tree can then be automatically refined using a given library of annotated attack trees.

In [5], Bryans et al. give a synthesis approach for sequential attack trees from a network model consisting of nodes and connectivity information and a library of attack tree templates. Both the network model and the library are constructed manually. In [33], Pinchinat et al. show that the complexity of the synthesis problem for sequential attack trees equipped with trace semantics is NP-complete. More specifically, their attack synthesis problem takes a trace and a library (basically a set of context-free grammar rules) as input and decides whether a sequential attack tree exists such that the given trace complies with the semantics (and provides the attack tree, too). An implementation is provided.

With one exception, all approaches rely on a library of already built models or another preexisting formal model of the system. The one approach uses all possible successful attacks, which are not commonly available in secure systems. We present a first approach to use a comparatively small amount of mostly non-attack data without needing preexisting system knowledge.

2 Background

2.1 Attack Trees

Attack-defense trees (for example, see Fig. 1) are labeled trees that allow refining major attack goals into smaller steps until no refinement is possible anymore. The atomic steps are called *basic events*. Each inner node of the tree represents a *composed event*. The operator attached to an inner node determines how its children need to succeed for the composed event to succeed as well.

In Fig. 1, we present an attack tree that models an attack on an IoT device from [3]. The attack tree represents the following attacks: The root represents

[3] We interpret the refinement specification as library similar to [33].

the goal of getting access to the IoT device and exploiting it. This goal is refined using the operator SAND, which requires all its inputs (in this case another composed event labeled with AND and the two basic events *run malicious script* and *exploit software vulnerability in IoT device*) to succeed in the given order (here, events need to succeed from left to right). The first input refines the intermediate goal *get access to IoT device* again into smaller steps. AND and OR here are the standard logical operations, i.e., AND represents a conjunctive and OR a disjunctive refinement. Thus, the leftmost subtree of the root specifies that the basic event *get credentials* needs to succeed for this subtree to succeed. Additionally, either the basic events *MAC address spoofing* and *finding a LAN access port* or the basic events *breaking WPA keys* and *finding WLAN access to the IoT device* need to succeed.

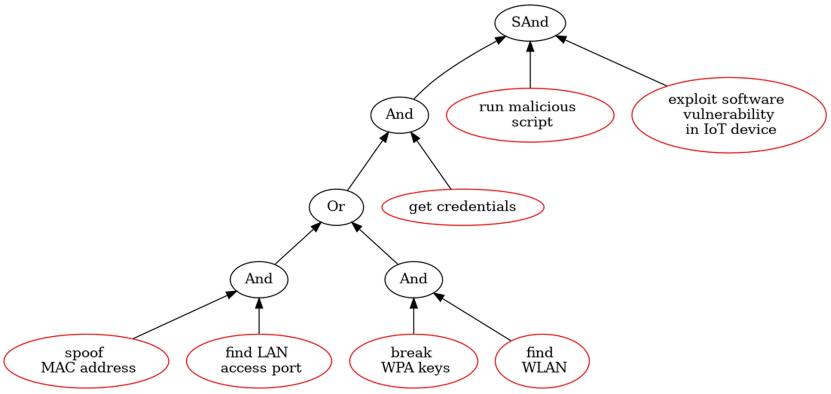


Fig. 1. A sequential attack tree from [3] representing an attacked IoT device.

We define the set of all operators $\mathsf{O} = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$ and use Definition 1 of [14] restricted to those operations. Intuitively, the operators are the standard logic operations. An *attack-defense tree* is a tuple $\mathsf{ADT} = (\mathsf{V}, \mathsf{E}, \mathsf{t})$, where (V, E) form a tree with all leaves being *basic events* (BE) and all inner nodes being *gates*. The gates are assigned their operator by the *type function* t.

A *trace* is a finite (ordered) sequence of basic events such that every basic event occurs at most once. The signature of a node v of an attack-defense tree ADT is the set of basic events that occur in the subtree rooted in v.

The level of detail introduced in this section is sufficient for the further development of this paper.

2.2 Attack Trees vs Decision Trees

We use attack trees and their extensions over decision trees, as decision trees are designed to generate high information gain and are thus likely to ignore very rare events. This is a problem especially concerning imbalanced data sets,

Table 1. The used hyperparameter in the algorithm with the abbreviations used in the paper.

Abbreviation	Description
ϵ	Number of new random models per generation
PSize	The size of the population preserved for the next generation
w_1	Weight for sensitivity in the fitness function
mutr	The ratio of mutations over crossovers
Act	Number of actions (e.g., mutations) per model in a generation
Gen	Number of generations until the algorithm terminates
trainr	Training rate. The portion of all traces in the training set

which we expect to deal with. It may be for instance solvable with specific sampling methods, but this step requires more work and is dependent on the context [8]. In addition, the graphical representation of decision trees might not be helpful for system designers to gain insights into the composition of successful and unsuccessful attacks since leaves are annotated by the result of the attack rather than the basic attack steps. Attack-defense trees allow synthesis based on sub-goals, naturally starting with the simplest action. Further, while decision trees can include information on time sequences, additional decision nodes are required for every relevant order of events. This results in a potential blowup of the model and a reduction in readability. Finally, attack-tree analysis can easily use information on already happened events to update the likelihood of successful attacks based on prior events. Thus, our models are valuable for the real-time surveillance of systems.

3 Genetic Algorithm for Learning *ADT*

This section presents our approach to learning attack trees from traces using genetic algorithms. In our genetic algorithm, the population of candidate solutions consists of attack-defense trees. Each attack-defense tree consists of composed and basic events, and the way they are linked are the chromosomes of the candidate attack-defense trees. In this section, we first define the fitness function to evaluate the fitness of every individual in the population. Then, we define mutation and crossovers. Note that the algorithm starts with completely random models and introduces a number ϵ of new random models to each generation to reduce the risk of premature convergence. An overview of all hyperparameters is listed in Table 1.

3.1 Fitness Function

To evaluate the fitness, we check how many log traces our attack-defense trees can correctly separate into successful or unsuccessful attacks. For a finite set $\mathcal{X}$,

we denote its size by $\|\mathcal{X}\|$. To this end, let $\mathsf{ADT} = (\mathsf{V}, \mathsf{E}, \mathsf{t})$ be an attack-defense tree, Tr be a set of traces and $\mathsf{c}\colon \mathsf{Tr} \to \{\mathsf{tt}, \mathsf{ff}\}$ be a labeling function, which assigns each trace its output value and $\mathsf{ADT}(\mathsf{tr})$ computes the label on a given ADT, where tt represents a successful attack and ff an unsuccessful attack. A straightforward definition of the fitness function using our labeled data set is the prediction accuracy of the attack-defense tree.

However, we assume that successful traces are less common than unsuccessful traces since, in a real-world application, one hopefully observes the system more often in its normal behavior than after a successful attack. Using the prediction accuracy as a fitness function in such a setting may result in an algorithm always returning false having a good accuracy. Therefore, we use a multi-objective approach including the *sensitivity* Sens and the *specificity* Spec of the prediction as follows:

$$\mathsf{Sens}(\mathsf{ADT}) = \frac{\#\mathsf{Tr}\text{ correctly labeled as }\mathsf{tt}}{\#\mathsf{Tr}\text{ in test set labeled as }\mathsf{tt}} = \frac{\|\{\mathsf{tr} \in \mathsf{Tr} \mid \mathsf{ADT}(\mathsf{tr}) = \mathsf{tt} \wedge \mathsf{c}(\mathsf{tr}) = \mathsf{tt}\}\|}{\|\{\mathsf{tr} \in \mathsf{Tr} \mid \mathsf{c}(\mathsf{tr}) = \mathsf{tt}\}\|}$$

$$\mathsf{Spec}(\mathsf{ADT}) = \frac{\#\mathsf{Tr}\text{ correctly labeled as }\mathsf{ff}}{\#\mathsf{Tr}\text{ in test set labeled as }\mathsf{ff}} = \frac{\|\{\mathsf{tr} \in \mathsf{Tr} \mid \mathsf{ADT}(\mathsf{tr}) = \mathsf{ff} \wedge \mathsf{c}(\mathsf{tr}) = \mathsf{ff}\}\|}{\|\{\mathsf{tr} \in \mathsf{Tr} \mid \mathsf{c}(\mathsf{tr}) = \mathsf{ff}\}\|}$$

Let $\mathsf{w}_1, \mathsf{w}_2$ be weights such that $0 \leq \mathsf{w}_1, \mathsf{w}_2 \leq 1; \mathsf{w}_1 + \mathsf{w}_2 = 1$. Then, we define the fitness function F for an attack-defense tree ADT as[4]

$$\mathsf{F}(\mathsf{ADT}) = \mathsf{w}_1 \cdot \mathsf{Sens}(\mathsf{ADT}) + \mathsf{w}_2 \cdot \mathsf{Spec}(\mathsf{ADT}).$$

This definition ensures F being smooth for

$$\|\{\mathsf{tr} \in \mathsf{Tr} \mid \mathsf{c}(\mathsf{tr}) = \mathsf{ff}\}\|, \|\{\mathsf{tr} \in \mathsf{Tr} \mid \mathsf{c}(\mathsf{tr}) = \mathsf{tt}\}\| \geq 1$$

Additionally, we define the *positive predictive value* PPV as the probability that for a trace tr and a prediction $\mathsf{ADT}(\mathsf{tr}) = \mathsf{tt}$ we actually have $\mathsf{c}(\mathsf{tr}) = \mathsf{tt}$.

$$\mathsf{PPV}(\mathsf{ADT}) = \frac{\|\{\mathsf{tr} \in \mathsf{Tr} \mid \mathsf{ADT}(\mathsf{tr}) = \mathsf{tt} = \mathsf{c}(\mathsf{tr})\}\|}{\|\{\mathsf{tr} \in \mathsf{Tr} \mid \mathsf{ADT}(\mathsf{tr}) = \mathsf{tt} = \mathsf{c}(\mathsf{tr}) \vee \mathsf{ADT}(\mathsf{tr}) = \mathsf{tt} \neq \mathsf{c}(\mathsf{tr})\}\|}.$$

If we report the fitness, the specificity, or the sensitivity, we use the result of the test set for each model to report the *mean* μ $\pm$ *standard deviation* σ according to their canonical definition for a set of models.

3.2 Mutations and Crossovers for Attack Trees

While mutations change labels of composed or basic events of the attack tree, crossovers change the position of subtrees within the tree. In this section, we restrict ourselves to attack trees, i.e., all inner nodes are either labeled with `AND` or `OR`. Our input data consists of a set of traces Tr, and their corresponding value is $\mathsf{c}\colon \mathsf{Tr} \to \{\mathsf{tt}, \mathsf{ff}\}$. We consider the following expectable *mutations*:

[4] There is no optimal combination for the weights. Hence, we explore different weights and how they influence the overall fitness in Sect. 4.

1. SWITCH to switch the position of any two basic events
2. CHANGE the label of an inner node v from AND to OR or vice versa

While mutations can be applied at every node of an attack tree, we have to ensure that *crossovers* do not violate the model's integrity, i.e., after a crossover, the model must still include every basic event exactly once. Thus, we only use swapping of subtrees containing the same basic events as crossover operations. Therefore, the amount of possible crossovers is smaller than that of mutations. This can result in no possible crossover between two models (except the root), especially in small models. In this case, the operation will be skipped.

3.3 Extension to Attack-Defense Trees

In contrast to many attack-defense tree models [24], we do not assign a player to inner nodes. Hence, we also cannot and do not assign a different semantics to these inner nodes based on the player controlling it.

To include defense actions in our learning approach, we only need to split the set of basic events BE into two disjoint sets (a set of basic events BE_A controlled by the attacker and a set of basic events BE_D controlled by the defender such that $\mathsf{BE} = \mathsf{BE}_\mathsf{A} \dot{\cup} \mathsf{BE}_\mathsf{D}$). Intuitively speaking, the borders of nodes in BE_D are colored green, while elements in BE_A have a red border. After learning the tree using our approach, we assign each basic event its respective player. The learning algorithm does not need to be changed; thus, the fitness does not change. Hence, we only perform experiments on attack trees.

4 Experiments

This section evaluates our approach experimentally. Firstly, we demonstrate that our algorithm is capable of synthesizing attack trees classifying all traces correctly for data sets containing all possible traces over a fixed set of basic events. Secondly, we run k-fold cross-validation to show that the selection of traces for the test and training sets does have a minor influence on the outcome. Thirdly, we compare our approach to logistic regression which allows us to assess the adequacy of our genetic algorithm in comparison to other learning approaches. Fourthly, we explore how to choose the weight parameters for specificity and sensitivity and how many successful traces are necessary to reach a high accuracy. Fifthly, we analyze different hyperparameter constellations to show their relevance to the outcome. Finally, we use the algorithm on the real KDD-Cup 1999 data set to demonstrate, that it could already be deployed for real-world systems. All experiments are executed on a machine with Intel ®Core ™i7 CPU and 15 GiB RAM running a VM image with 4 GiB RAM and one Core.[5] If not stated otherwise, the algorithm can use the operators AND, OR and NOT.

[5] The artifact can be found at https://doi.org/10.5281/zenodo.8352279.

4.1 Generating Input Datasets

Generating Data. Let $\mathsf{ADT} = (\mathsf{V}, \mathsf{E}, \mathsf{t})$ be an attack-defense tree over the set of basic events BE. We generate traces of basic events randomly (without using any information of the structure of the tree) and then, compute the valuation of the trace w.r.t. the attack-defense tree $\mathsf{ADT}(\mathsf{tr})$ to obtain the label function $\mathsf{c} : \mathsf{Tr} \to \{\mathsf{tt}, \mathsf{ff}\}$. A trace ends if all be were attempted or if the root of the tree evaluates to tt.

Each basic event can only occur once in a trace. Hence, for an *ADT* with n basic events there are $\frac{n!}{(n-m)!}$ traces of length m. So, there is a total of $\sum_{m=1}^{n} \frac{n!}{(n-m)!}$ traces[6].

Since the number of possible traces grows rapidly and in a real-world system, we may also only have a limited amount of data available, we restrict ourselves to a fixed number of traces in each experiment.

Splitting Data. Given a set of traces Tr and a corresponding labeling function c the dataset is split into a training and a test set. Our algorithm only uses the training set to learn an attack tree. The test set is used for the final evaluation of the fitness of the learned model. The sets are split with the *training rate* trainr preserving the ratio between tt and ff labeled traces.

4.2 Sanity Check: Perfect Fitting for Small Models

First, we demonstrate that our algorithm is potentially able to lead to an optimal outcome w.r.t. the given fitness function F_c (see Subsect. 3.1).

Setup. We use 80 randomly generated models with nine nodes and five basic events each. All inner nodes are labeled with either `AND` or `OR` (i.e. we restrict ourselves to attack trees here). We use all existing traces for these models as input for both training and testing. While this setup results in overfitting the dataset, this is acceptable in this situation since we know there are no further traces (not contained in the dataset). We use the following configuration for the hyperparameters: number of generations $\mathsf{Gen} = 120$, population size $\mathsf{PSize} = 140$, mutation rate $\mathsf{mutr} = 0.8$, and number of actions $\mathsf{Act} = 1$. Act is set to 1 to get an intuition on how many actions on the models are needed for the result.

Results. All models were learned with 100 % accuracy in the given generations limit. The mean time until termination was 2.5 s.

Observation 1. *Small models can be learned to 100% accuracy if all existing traces are given as input.*

The high number of iterations needed to achieve the goal suggests that the genetic algorithm is able to learn the corresponding models, but may not be the best choice to learn accurate attack-defense trees if the dataset is small enough

[6] We already have about one million distinct traces with $n = 9$. However, this is only an upper bound since we stop the traces as soon as the root turns tt.

to apply exact solutions. This is due to the unsystematic search. For instance, a mutation can simply undo the previous mutation, which will keep the better model unchanged for the next generation. Thus, we are not guaranteed to make progress during a generation.

4.3 Crossvalidation: No Overfitting of Models

Setup. To further validate the algorithm, we perform k-fold crossvalidation with $k = 10$, i.e. training the model on nine of ten random subsets and validating it on the remaining one. To check if the algorithm is over fitting the data, we used 35 attack trees and generated a training set of 800 traces on each. The experiment used the following hyperparameters: number of generations $\mathsf{Gen} = 60$, population size $\mathsf{PSize} = 40$, mutation rate $\mathsf{mutr} = 0.8$, number of actions $\mathsf{Act} = 3$.

Results. The cross validation showed only minor deviations between the folds and the performance on the evaluation set, i.e. the left out fold. Overall the mean of the standard deviation in Fitness over all datasets was 0.003 ± 0.002.

Observation 2. *Our genetic algorithm is, in this setting, robust to overfitting due to sampling.*

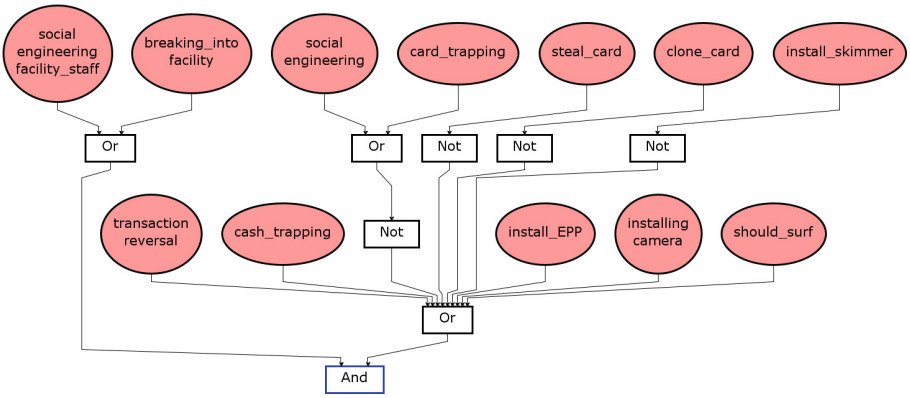
4.4 Comparison to Logistic Regression

Attack trees can be seen as *Boolean functions*. *Logistic Regression* is a viable competitor for learning Boolean functions since it is optimized to predict binary endpoints from labeled data sets [16].

Setup. To be able to compare both approaches, we chose datasets consisting of 1000 traces in a way that about $50 \pm 1\%$ of the traces are labeled with tt. This was done to have a balanced relation in accuracy. Having 96% ff labeled traces as in Subsect. 4.5 in these experiments leads to classifiers always returning false having an accuracy of 96%, which does not provide any information on attacks. Due to the high frequency of attacks, this setup is unlikely in a real-world scenario. We use setups with fewer attacks in the following sections.

Creating equally sized sets of tt and ff labeled traces implies the necessity to choose an attack tree with a mix of both operators since more nodes labeled with AND reduce the quantity of possible tt labeled traces and nodes labeled with OR cause the opposite. Thus, for these experiments, we selected only attack trees of which the algorithm was able to produce enough traces for both labels.

The fitness function for our genetic algorithm was set to the overall accuracy (i.e. F_c in Subsect. 3.1). We learned the logistic LR using R version 3.6.3 and indicator variables for every basic event showing if the event is present in a trace. In attack trees, the order of basic events does not matter. Hence, we only allowed traces that could not be reordered in each other (i.e. no permutations). Overall, we used 35 attack trees ranging from 31 to 53 nodes (and 16 to 26 basic events). We used the following configuration for the hyperparameters: number of generations $\mathsf{Gen} = 60$, population size $\mathsf{PSize} = 40$, mutation rate $\mathsf{mutr} = 0.8$, number of actions $\mathsf{Act} = 3$ and training rate $\mathsf{trainr} = 0.8$.



(a) Genetic algorithm (Fitness 0.97)

$$p(X) = \frac{1}{1 + e^{-(-5.66869 + X * \beta)}} \tag{1}$$

$$\beta^{-1} = (\, 1.38 \;\; 0.44 \;\; 0.48 \;\; 0.44 \;\; 0.05 \;\; 0.17 \;\; 0.74 \;\; 0.63 \;\; 0.90 \;\; 4.67 \;\; 4.61 \;\; 1.53 \,) \tag{2}$$

(b) Logistic regression (Fitness 0.95)

Fig. 2. The resulting models for the tree from [7]. The model in (a) was simplified manually, by collapsing subsequent operators of the same kind into one. The equation in (b) displays the corresponding logistic regression model.

Results. The accuracy is measured according to the fitness function F_c (see Subsect. 3.1) for every single of the 35 models on the test set containing 200 traces and after training on 800 traces each.

Observation 3. *Our genetic algorithm approach performs comparable to logistic regression with a mean difference of only 0.04 (standard deviation 0.01) but, in contrast, results in a graphical, interpretable model.*

In more detail, our approach yielded a mean accuracy of 0.82 (with a standard deviation 0.08), and the logistic regression produced 0.86±0.08 (mean±standard deviation). We show a real-world example in Fig. 2. The slight inferiority of our algorithm is due to logistic regression being optimized for this specific setting. However, it is only a suitable competitor in this specific setting. An expansion to non-binary labels, e.g. undecided for an unknown system status, would make a comparison with multi-label classification algorithms necessary. Also, results will differ in specific fields or data settings. Additionally, introducing time-dependent features results in a massive increase in independent variables, which results in a higher standard error and unstable results [16]. Furthermore, logistic regression obviously does not directly create an attack tree, which is one main goal of this paper. Decision trees as an alternative have been discussed in Subsect. 2.2.

4.5 Learning Attack Trees

Here we analyze the performance of the genetic algorithm for attack trees (see Subsect. 3.2).

Setup. We use 35 datasets as described in Subsect. 4.4 using subsets with different ratios of traces labeled as unsuccessful to successful in our experiments. Our goal is to estimate which proportion of traces labeled tt is needed to generate fit models in real-world systems. More specifically, we create datasets with 1000 traces where 5, 10, 20, 40, 80, 160, 320, or 500 of the 1000 overall traces are labeled tt. For each dataset, we experiment with weight w_1 from 0.1 to 0.9 in steps of size 0.1 (and set $\mathsf{w}_2 = 1 - \mathsf{w}_1$ accordingly). The other parameters are equal to the last experiment, i.e. number of generations $\mathsf{Gen} = 60$, population size $\mathsf{PSize} = 40$, mutation rate $\mathsf{mutr} = 0.8$, number of actions $\mathsf{Act} = 3$ and training rate $\mathsf{trainr} = 0.8$.

Results. We report the sensitivity and specificity of all final models on the test set in a 3D-Plot[7]. Parts of the results are listed in Fig. 3.

Observation 4. *Only about 4% of all traces in the dataset need to be successful attacks to learn fit functions with a mean sensitivity of up to 95%, or specificity of up to 99%.*

The mean of sensitivity and specificity show a positive correlation to their weights with *Pearson's correlation coefficient* $r_{Pearson} = 0.778$ ($p :< 0.001$) and $r_{Pearson} = 0.887$ ($p :< 0.001$) for sensitivity and specificity, respectively. Additionally, there is no significant correlation between sensitivity and the number of tt traces with $r_{Pearson} = 0.317$ ($p : 0.007$), the same seems to be true for specificity $r_{Pearson} = -0.200$ ($p : 0.092$). These results suggest that the impact of choosing the weight is bigger than of choosing the number of successful attack traces.

This seems promising for real-world scenarios since there may only be small amount of successful attacks but an abundant amount of unsuccessful or incomplete ones to learn attack trees from. In terms of runtime, it took about 10–12 minutes for each execution of the algorithm on all models. The average execution time for a single prompt was about 20–30 s. More information on runtimes can be found at Subsect. 4.6.

Our results show that the weights of the sensitivity and the specificity need to be chosen appropriately within the application scenario. A high-risk security infrastructure might rely on a high sensitivity to spot all ongoing attacks. On the other hand, a low specificity may lead to many false positives, especially, if we have only a small number of attack traces. For instance, with 20 attack traces (traces labeled tt), we have a sensitivity of 82.7% and specificity of 67.2% at $\mathsf{w}_1 = 0.7$. Given this information, that means that there were 321 false positives per 1000 traces and a positive predictive value PPV of about 4.9%. Hence, 95.1%

[7] https://www.model.in.tum.de/~kraemerj/upload/3dplots/.

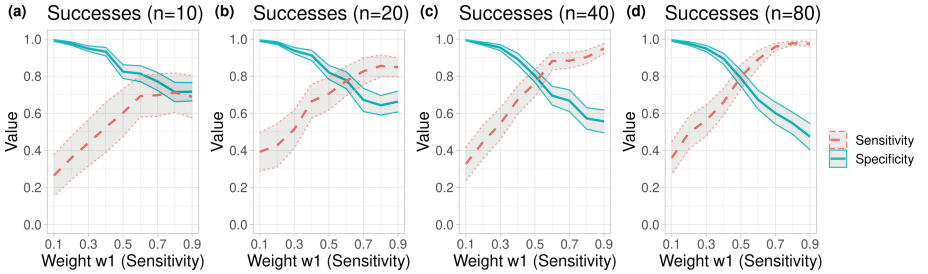


Fig. 3. Sensitivity and specificity are depicted as mean with 95% confidence interval depending on the weight w_1 for sensitivity in the fitness function. n specifies the number of successful attacks among the 1000 traces in the dataset.

of all predictions on our full dataset are false alarms. Favoring specificity with $\mathsf{w}_1 = 0.3$ would result in a PPV of about 18.6%.

Therefore, we cannot give a general recommendation regarding the weight w_1, as it is highly dependent on the security needs and possible damages.

4.6 Hyperparameter Optimization

The previous sections were dedicated to optimizing the parameters of the weight w and the number of attack traces n. In this section, we optimize the parameters inherent to genetic algorithms: the number of generations Gen, the population size PSize, the mutation rate mutr and the number of actions Act.

Setup. We learn attack trees from the 35 datasets from Subsect. 4.5 on a fixed set of hyperparameters only changing one parameter at once. Each parameter was set to multiple values while both sensitivity and specificity were recorded as dependent variables. The base setting was $n = 80$, $\mathsf{w}_1 = 0.5$, Gen $= 60$, PSize $= 40$, mutr $= 0.8$, Act $= 3$ and trainr $= 0.8$.

For line-searching Gen We used metadata provided by the previous experiment Subsect. 4.5. It contains the fitness of each model after each generation. Thus, we used 60 data points each. For search the PSize was altered in steps of 10 between 10 and 100. The mutr was between 0.1 and 0.9 and increased in steps of 0.1. Lastly, Act was checked in a range from 1 up to 10 increasing in steps of 1.

We created a mixed linear model as described in [4]. This model is needed as performance varies depending on the dataset and therefore, the different settings cannot be treated as independent.

Results. Estimates for the mixed linear model after altering one parameter of the standard setup are shown in Table 2. All given runtime Data below is with regard to a full run over all 35 models. Except for the number of generations Gen (and for specificity Spec on the number of actions Act), none of the results is statistically significant, leading to:

Table 2. Slopes in the mixed linear model after altering one parameter of the standard setup. Except for the estimates for the number of generations Gen and for the specificity Spec on the number of actions Act, none of the results is statistically significant suggesting a low impact on algorithm performance.

	Estimate for Sens	Estimate for Spec
# Generations	0.002 ($t: < 0.001$)	< 0.001 ($t: < 0.001$)
Population Size	< 0.001 ($t:$ 0.882)	< 0.001 ($t:$ 0.354)
Mutation Rate	< 0.001 ($t:$ 0.717)	0.014 ($t:$ 0.560)
# Actions	0.004 ($t:$ 0.268)	< 0.001 ($t:$ 0.955)

Observation 5. *Choosing a sufficiently high number of generations suffices for a good learning result in our experiments. The other hyperparameters only have a minor impact on the fitness of the learned models.*

4.7 Real-World Demonstration: KDD Cup 1999 Dataset

Finally, we validate our algorithm on the *KDD Cup 1999 Dataset*[8] generated in a simulated military network environment. In this dataset, every entry denotes a summary of one single connection to a server, which is either an intrusion or a normal connection. The dataset consists of 4.9 and 0.3 million entries in the training and test set, respectively. Our goal is to learn an attack-defense tree to distinguish intrusions from normal connections. Note in contrast to previous experiments, no attack tree is used to generate the traces.

Setup. We interpret every column as a single event. However, some columns in the dataset are continuous. Thus, we split the values at a given *quantile* q and see every value equal or greater than it as an attempted event of this column. Additionally, our algorithm currently only supports attack-defense trees using a single root node. Hence, we change the different attack labels e.g. *guess_passwd* to one single label. The label is tt if any of the attack labels is tt otherwise we interpret the trace as an unsuccessful attack. We use the hyperparameters Gen $= 100$, PSize $= 60$, mutr $= 0.8$, Act $= 3$. Weight w_1 and quantile q are changed by 0.1 between 0.1 and 0.9 in every run. We then record the best resulting model and the corresponding Sens and Spec.

Results. We present the results for the best models in Fig. 4. Please note that the fallout is simply $1 - \text{Spec}$. The point with the lowest Euclidean distance to the optimum, i.e. $(0, 1)$, is the combination of $w_1 = 0.9$ and $q = 0.4$ with Sens $= 0.968$ and Spec $= 0.994$ on the test set. Interestingly, the high values of weight w_1 produced the best result, although only 19.9% of the entries are normal. We suspect this may be due to the higher stability of the specificity Spec reported

[8] https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

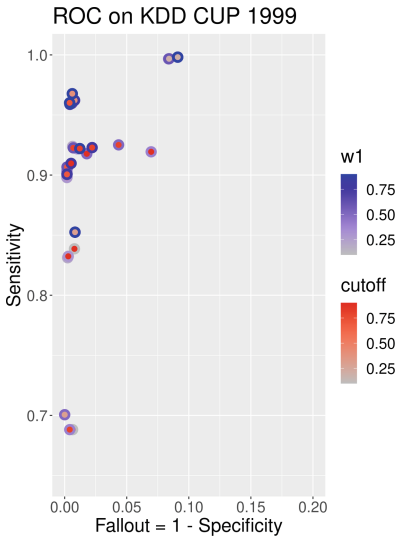


Fig. 4. ROC-Curve of setups with w_1 and q ranging between 0.1 and 0.9. 14 points with $\mathsf{Spec} < 0.2$ *or* $\mathsf{Sens} < 0.5$ were removed for better readability. The best point uses $\mathsf{w}_1 = 0.9$ and $\mathsf{q} = 0.4$ resulting in $\mathsf{Sens} = 0.968$ and $\mathsf{Spec} = 0.994$.

in the earlier experiments. While the Performance of our model is similar to the one reported by [1] using a decision-tree-based approach, our setup uses a less complex set of labels than their approach does.

5 Conclusion and Future Work

We have presented the first algorithm to learn attack trees from sets of traces labeled as successful or unsuccessful attacks. To this end, we have used genetic algorithms. We consider this a step forward to *bridging the gap* between real-world applications and their respective analysis since such traces are often collected anyway in many application domains in the form of logs. Consequently, the attack trees can be created automatically, eliminating the burden of manual involvement of domain experts in other approaches. Our experiments have shown that (i) the accuracy of our graphical models is comparable to (non-graphical) logistic regression while (ii) producing **small and straightforward graphical models**, and (iii) that only a **small amount of attack traces** among all traces are necessary to learn accurate models. Additionally, the algorithm was able to classify data from the preexisting KDD CUP 1999 data set with a performance that is comparable with existing models but in contrast with **no human intervention** required. Surprisingly, compared to the previous approaches on fault trees, our algorithms seem to need far fewer traces to generate usable models.

Future Work. Our algorithm can be extended in various ways. Firstly, it would be worthwhile to test whether including prior knowledge in generation 0 leads

to fitter models. Secondly, one can experiment with richer sets of mutations and crossovers to include domain knowledge or existing attack tree models. Thirdly, including additional types of inner nodes could overcome the condition of a basic event only occurring once per trace. Using more real-world data the choice of hyperparameters could be better justified or tested with different tuners. Additionally, extensions to the algorithm like using multi-objective approaches for fitness, may improve the performance. Setting a baseline for future solutions with this algorithm may help compare this approach with options from other fields e.g. SMT.

Acknowledgement. The work was partially supported by the MUNI Award in Science and Humanities (MUNI/I/1757/2021) of the Grant Agency of Masaryk University.

References

1. Jalil, K.A., Kamarudin, M.H., Masrek, M.N.: Comparison of machine learning algorithms performance in detecting network intrusion. In: 2010 International Conference on Networking and Information Technology, pp. 221–226. IEEE (2010)
2. Alhomidi, M., Reed, M.: Finding the minimum cut set in attack graphs using genetic algorithms. In: 2013 International Conference on Computer Applications Technology (ICCAT), pp. 1–6. IEEE (2013)
3. André, É., et al.: Parametric analyses of attack-fault trees. In: 2019 19th International Conference on Application of Concurrency to System Design (ACSD), pp. 33–42. IEEE (2019)
4. Bates, D., et al.: Fitting linear mixed-effects models using lme4 (2014)
5. Bryans, J., et al.: A template-based method for the generation of attack trees. In: Laurent, M., Giannetsos, T. (eds.) WISTP 2019. LNCS, vol. 12024, pp. 155–165. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41702-4_10
6. Budde, C.E., Bucur, D., Verkuil, B.: Automated fault tree learning from continuous-valued sensor data. Int. J. Prognostics Health Manag. **13**(2) (2022). https://doi.org/10.36001/ijphm.2022.v13i2.3160. ISSN 2153-2648
7. Buldas, A., et al.: Attribute evaluation on attack trees with incomplete information. Comput. Secur. **88**, 101630 (2020)
8. Chawla, N.V.: C4. 5 and imbalanced data sets: investigating the effect of sampling method, probabilistic estimate, and decision tree structure. In: Proceedings of the ICML, Toronto, ON, Canada, vol. 3, p. 66. CIBC (2003)
9. Fila, B., Wideł, W.: Attack–defense trees for abusing optical power meters: a case study and the OSEAD tool experience report. In: Albanese, M., Horne, R., Probst, C.W. (eds.) GraMSec 2019. LNCS, vol. 11720, pp. 95–125. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-36537-0_6
10. Gadyatskaya, O., Trujillo-Rasua, R.: New directions in attack tree research: catching up with industrial needs. In: Liu, P., Mauw, S., Stølen, K. (eds.) GraMSec 2017. LNCS, vol. 10744, pp. 115–126. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74860-3_9
11. Gadyatskaya, O., et al.: Attack trees for practical security assessment: ranking of attack scenarios with ADTool 2.0. In: Agha, G., Van Houdt, B. (eds.) QEST 2016. LNCS, vol. 9826, pp. 159–162. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43425-4_10

12. Gonçalves, E.C., Freitas, A.A., Plastino, A.: A survey of genetic algorithms for multi-label classification. In: 2018 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2018)
13. Gupta, M., et al.: Matching information security vulnerabilities to organizational security profiles: a genetic algorithm approach. Decis. Support Syst. **41**(3), 592–603 (2006)
14. Hermanns, H., et al.: The value of attack-defence diagrams. In: Piessens, F., Viganò, L. (eds.) POST 2016. LNCS, vol. 9635, pp. 163–185. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49635-0_9
15. Hong, J.B., Kim, D.S., Takaoka, T.: Scalable attack representation model using logic reduction techniques. In: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, pp. 404–411. IEEE (2013)
16. Hosmer, D.W., Jr., Lemeshow, S., Sturdivant, R.X.: Applied Logistic Regression, vol. 398. Wiley, Hoboken (2013)
17. Ivanova, M.G., et al.: Attack tree generation by policy invalidation. In: Akram, R.N., Jajodia, S. (eds.) WISTP 2015. LNCS, vol. 9311, pp. 249–259. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24018-3_16
18. Jhawar, R., et al.: Attack trees with sequential conjunction. In: Federrath, H., Gollmann, D. (eds.) SEC 2015. IAICT, vol. 455, pp. 339–353. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18467-8_23
19. Jhawar, R., et al.: Semi-automatically augmenting attack trees using an annotated attack tree library. In: Katsikas, S.K., Alcaraz, C. (eds.) STM 2018. LNCS, vol. 11091, pp. 85–101. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01141-3_6
20. Jimenez-Roa, L.A., et al.: Automatic inference of fault tree models via multi-objective evolutionary algorithms. IEEE Trans. Dependable Secure Comput. **20**(4), 3317–3327 (2023). https://doi.org/10.1109/tdsc.2022.3203805. ISSN 1545-5971
21. Jürgenson, A., Willemson, J.: On fast and approximate attack tree computations. In: Kwak, J., Deng, R.H., Won, Y., Wang, G. (eds.) ISPEC 2010. LNCS, vol. 6047, pp. 56–66. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12827-1_5
22. Kim, D., Choi, J., Han, K.: Risk management-based security evaluation model for telemedicine systems. BMC Med. Inform. Decis. Mak. **20**(1), 1–14 (2020)
23. Kordy, B., Pietre-Cambacedes, L., Schweitzer, P.: DAG-based attack and defense modeling: don't miss the forest for the attack trees. CoRR, abs/1303.7397 (2013). http://arxiv.org/abs/1303.7397
24. Kordy, B., et al.: Foundations of attack-defense trees. In: Degano, P., Etalle, S., Guttman, J. (eds.) FAST 2010. LNCS, vol. 6561, pp. 80–95. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19751-2_6 ISBN 978-3-642-19750-5
25. Kumar, R., Stoelinga, M.: Quantitative security and safety analysis with attack-fault trees. In: High Assurance Systems Engineering (HASE), pp. 25–32 (2017). https://doi.org/10.1109/HASE.2017.12
26. Lenin, A., Willemson, J., Sari, D.P.: Attacker profiling in quantitative security assessment based on attack trees. In: Bernsmed, K., Fischer-Hübner, S. (eds.) NordSec 2014. LNCS, vol. 8788, pp. 199–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11599-3_12
27. Linard, A., Bucur, D., Stoelinga, M.: Fault trees from data: efficient learning with an evolutionary algorithm. In: Guan, N., Katoen, J.-P., Sun, J. (eds.) SETTA 2019. LNCS, vol. 11951, pp. 19–37. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-35540-1_2

28. Majeed, P.G., Kumar, S.: Genetic algorithms in intrusion detection systems: a survey. Int. J. Innov. Appl. Stud. **5**(3), 233 (2014)
29. RTO NATO. Improving common security risk analysis. Technical report, RTO Technical Report TR-IST-049, Research and Technology Organisation of NATO (2008)
30. Pawar, S.N.: Intrusion detection in computer network using genetic algorithm approach: a survey. Int. J. Adv. Eng. Technol. **6**(2), 730 (2013)
31. Pinchinat, S., Acher, M., Vojtisek, D.: ATSyRa: an integrated environment for synthesizing attack trees. In: Mauw, S., Kordy, B., Jajodia, S. (eds.) GraMSec 2015. LNCS, vol. 9390, pp. 97–101. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29968-6_7
32. Pinchinat, S., Acher, M., Vojtisek, D.: Towards synthesis of attack trees for supporting computer-aided risk analysis. In: Canal, C., Idani, A. (eds.) SEFM 2014. LNCS, vol. 8938, pp. 363–375. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15201-1_24
33. Pinchinat, S., Schwarzentruber, F., Lê Cong, S.: Library-based attack tree synthesis. In: Eades III, H., Gadyatskaya, O. (eds.) GraMSec 2020. LNCS, vol. 12419, pp. 24–44. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62230-5_2
34. Ramos, J.L.H., Skarmeta, A.: Assessing vulnerabilities in IoT-based ambient assisted living systems. Secur. Privacy Internet Things Challenges Solutions **27**, 94 (2020)
35. Rosmansyah, Y., Hendarto, I., Pratama, D.: Impersonation attack-defense tree. Int. J. Emerg. Technol. Learn. (iJET) **15**(19), 239–246 (2020)
36. Schneier, B.: Secrets & Lies: Digital Security in a Networked World, 1st edn. Wiley, New York (2000). ISBN 0471253111
37. Sheyner, O., et al.: Automated generation and analysis of attack graphs. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, SP 2002, Washington, DC, USA, p. 273. IEEE Computer Society (2002). http://dl.acm.org/citation.cfm?id=829514.830526. ISBN 0-7695-1543-6
38. Shostack, A.: Threat Modeling: Designing for Security. Wiley, Hoboken (2014)
39. Vigo, R., Nielson, F., Nielson, H.R.: Automated generation of attack trees. In: 2014 IEEE 27th Computer Security Foundations Symposium, pp. 337–350. IEEE (2014)
40. Widel, W., et al.: Beyond 2014: formal methods for attack tree-based security modeling. ACM Comput. Surv. (CSUR) **52**(4), 1–36 (2019)