On Fast and Approximate Attack Tree Computations

Aivo Jürgenson[1,2], Jan Willemson[3]

[1] Tallinn University of Technology, Raja 15,Tallinn 12618, Estonia
aivo.jurgenson@eesti.ee
[2] Cybernetica, Akadeemia 21, Tallinn 12618, Estonia
[3] Cybernetica, Aleksandri 8a, Tartu 51004, Estonia
jan.willemson@gmail.com

Abstract. In this paper we address the problem of inefficiency of exact attack tree computations. We propose several implementation-level optimizations and introduce a genetic algorithm for fast approximate computations. Our experiments show that for attack trees having less than 30 leaves, the confidence level of 89% can be achieved within 2 seconds using this algorithm. The approximation scales very well and attack trees of practical size (up to 100 leaves) can be analyzed within a few minutes.

1 Introduction

Structural methods for security assessment have been used for several decades already. Called *fault trees* and applied to analyse general security-critical systems in early 1980-s [1], they were adjusted for information systems and called *threat logic trees* by Weiss in 1991 [2]. In the late 1990-s, the method was popularized by Schneier under the name *attack trees* [3]. Since then, it has evolved in different directions and has been used to analyze the security of several practical applications, including PGP [4], Border Gateway Protocol [5], SCADA systems [6], e-voting systems [7], etc. We refer to [8, 9] for good overviews on the development and applications of the methodology.

Even though already Weiss [2] realized that the attack components may have several parameters in practice, early studies mostly focused on attack trees as a mere attack dependence description tool and were limited to considering at most one parameter at a time [3, 10, 11]. A substantial step forward was taken by Buldas *et al.* [12] who introduced the idea of game-theoretic modeling of the attacker's decision making process based on several interconnected parameters like the cost, risks and penalties associated with different elementary attacks. This approach was later

refined by Jürgenson and Willemson by first extending the parameter domain from point values to interval estimates [13] and then by creating the first semantics for multi-parameter attack trees, consistent with the general framework of Mauw and Oostdijk [11, 14].

Even though being theoretically sound, the results of Jürgenson and Willemson are rather discouraging from an engineering point of view. Even with all the optimizations proposed in [14], they are still able to analyze the trees of at most 20 leaves in reasonable time and this may not suffice for many practical applications. Hence, the aim of this paper is to improve their results in two directions. First, we implement several additional optimizations and second, we create and test a genetic algorithm for fast approximations.

The paper is organized as follows. First, in Section 2 we will briefly define attack trees and the required parameters. Then Section 3 will explain our new set of optimizations, which in turn will be tested for performance in Section 4. Section 5 will cover our genetic algorithm and finally Section 6 will draw some conclusions.

2 Attack Trees

Basic idea of the attack tree approach is simple – the analysis begins by identifying one *primary threat* and continues by dividing the threat into subattacks, either all or some of them being necessary to materialize the primary threat. The subattacks can be divided further etc., until we reach the state where it does not make sense to divide the resulting attacks any more; these kinds of non-splittable attacks are called *elementary attacks* and the security analyst will have to evaluate them somehow.

During the splitting process, a tree is formed having the primary threat in its root and elementary attacks in its leaves. Using the structure of the tree and the estimations of the leaves, it is then (hopefully) possible to give some estimations of the root node as well. In practice, it mostly turns out to be sufficient to consider only two kinds of splits in the internal nodes of the tree, giving rise to AND- and OR-nodes. As a result, an AND-OR-tree is obtained, forming the basis of the subsequent analysis. We will later identify the tree as a (monotone) Boolean formula built on the set of elementary attacks as literals.

The crucial contribution of Buldas *et al.* [12] was the introduction of four game-theoretically motivated parameters for each leaf node of the tree. This approach was later optimized in [14], where the authors concluded that only two parameters suffice. Following their approach, we

consider the set of elementary attacks $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ and give each one of them two parameters:

- p_i – success probability of the attack X_i,
- $\mathsf{Expenses}_i$ – expected expenses (i.e. costs plus expected penalties) of the attack X_i.

Besides these parameters, there is a global value Gains expressing the benefit of the attacker if he is able to materialize the primary threat.

3 Efficient Attack Tree Computations

Let us have the attack tree expressed by the monotone Boolean formula $\mathcal{F}$ built on the set of elementary attacks $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$. In the model of [14], the expected outcome of the attacker is computed by maximizing the expression

$$\mathsf{Outcome}_\sigma = p_\sigma \cdot \mathsf{Gains} - \sum_{X_i \in \sigma} \mathsf{Expenses}_i \tag{1}$$

over all the assignments $\sigma \subseteq \mathcal{X}$ that make the Boolean formula $\mathcal{F}$ true. Here p_σ denotes the success probability of the primary threat and as shown in [14], this quantity can be computed in time linear in the number n of elementary attacks. The real complexity of maximizing (1) comes from the need to go through potentially all the 2^n subsets $\sigma \subseteq \mathcal{X}$. Of course, there are some useful observations to make.

- The Boolean function $\mathcal{F}$ is monotone and we are only interested in the satisfying assignments σ. Hence, it is not necessary to consider subsets of non-satisfying assignments.
- In [14], Theorem 1, it was proven that if for some AND-node in the attack tree the assignment σ evaluates some of its children as true and others as false, this σ can be disregarded without affecting the correct outcome.

We start the contributions of the current paper by additionally noting that the DPLL algorithm [15], used in [14] to generate all the satisfying assignments, introduces a lot of unnecessary overhead. The formula $\mathcal{F}$ first needs to be transformed to CNF and later maintained as a set of clauses, which, in turn, are sets of literals. Since set is a very inconvenient data structure to handle in the computer, we can hope for some performance increase by dropping it in favor of something more efficient.

In our new implementation, we keep the formula $\mathcal{F}$ as it is – in the form of a tree. The assignments σ are stored as sequences of ternary bits, i.e. strings of three possible values t, f and u (standing for true, false and $\mathsf{undefined}$, respectively). The computation rules of the corresponding ternary logic are natural, see Table 1.

$\&$	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

$\vee$	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

Table 1. Computation rules for ternary logic

In its core, our new algorithm still follows the approach of DPLL – we start off with the assignment $[\mathsf{u}, \mathsf{u}, \ldots, \mathsf{u}]$ and proceed by successively trying to evaluate the literals as f and t. Whenever we reach the value t for the formula $\mathcal{F}$, we know that all the remaining u-values may be arbitrary and it is not necessary to take the recursion any deeper. Similarly, when we obtain the value f for the formula $\mathcal{F}$, we know that no assignment of u-values can make $\mathcal{F}$ valid. Thus the only case where we need to continue recursively, is when we have $\mathcal{F} = \mathsf{u}$. This way we obtain Algorithm 1, triggered by `process_satisfying_assignments`$([\mathsf{u}, \mathsf{u}, \ldots, \mathsf{u}])$.

Algorithm 1 Finding the satisfying assignments

Require: Boolean formula $\mathcal{F}$ corresponding to the given AND-OR-tree
1: **Procedure** `process_satisfying_assignments`(σ)
2: Evaluate $\mathcal{F}(\sigma)$
3: **if** $\mathcal{F}(\sigma) = \mathsf{f}$ **then**
4: Return;
5: **end if**
6: **if** $\mathcal{F}(\sigma) = \mathsf{t}$ **then**
7: Output all the assignments obtained from σ by setting all its u-values to t and f in all the possible ways;
Return;
8: **end if**
9: //reaching here we know that $\mathcal{F}(\sigma) = \mathsf{u}$
10: Choose X_i such that $\sigma(X_i) = \mathsf{u}$
11: `process_satisfying_assignments`$(\sigma/[X_i := \mathsf{f}])$;
12: `process_satisfying_assignments`$(\sigma/[X_i := \mathsf{t}])$;

Even though being conceptually simple, Algorithm 1 contains several hidden options for optimization. The first step to pay attention to lies already in line 2, the evaluation of $\mathcal{F}(\sigma)$. The evaluation process naturally follows the tree structure of $\mathcal{F}$, moving from the leaves to the root using the computation rules given by Table 1. However, taking into account Theorem 1 of [14] (see the second observation above), we can conclude that whenever we encounter a node requiring evaluation of t&f or f&t, we can abort this branch of the recursion immediately, since there is a global optimum outcome in some other branch.

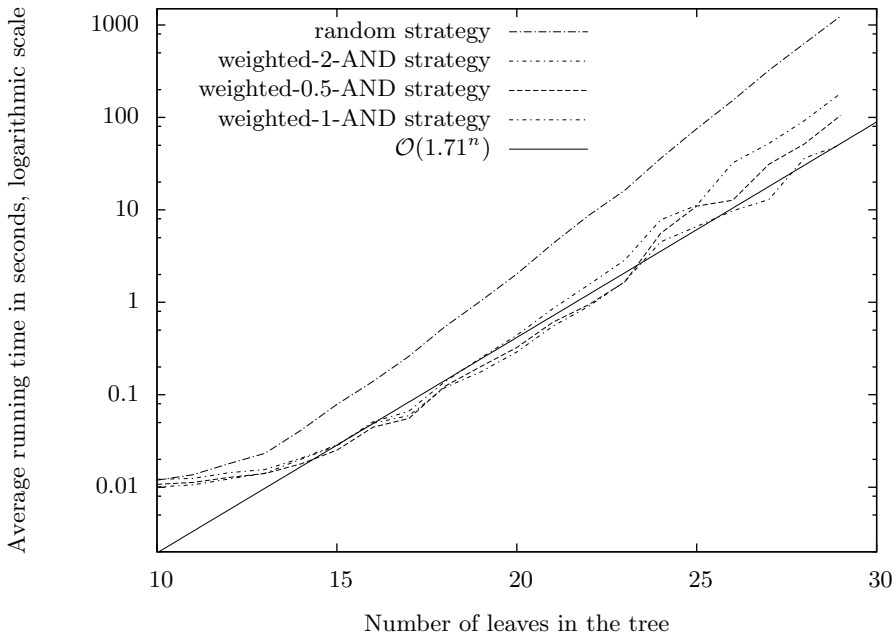
In the implementation, this kind of exception to evaluation is modelled as additional data type shortcut-false to the already existing true, false and undefined Boolean types. If the situation is encountered during the recursive evaluation of $\mathcal{F}$, shortcut-false is returned immediately to the highest level and the entire branch of processing is dropped.

Another, somewhat more obvious optimization lies within the line 10 of Algorithm 1. Of course it would be the simplest to pick the next undefined literal randomly (or in some predefined order, which gives the same result for randomly-generated test trees). However, intuitively this approach is one of the worst possible, since the working time of the algorithm depends on the number of the generated satisfying assignments. Hence, the algorithm will be faster if we can systematically disregard larger recursive branches. This is the reason why we first assign undefined literals as f on line 11 and check first, if the whole formula has become non-satisfied.

Still, a clever choice of the order of the undefined literals to specify can speed up this process even further. We implemented and tested several possible strategies.

1. Random – the next undefined literal is picked randomly.
2. Most-AND – for each undefined literal we compute the number of AND-nodes on the path from the corresponding leaf to the root and pick the onest with the highest score first.
3. Weighted-AND – the ordering routine is similar to Most-AND, but all the AND-nodes on the path do not have an equal weight. The intuition behind this approach is that when we can exclude a larger subtree, we should be able to cut off more hopeless recursion branches as well, hence it makes more sense to prefer paths with AND-nodes closer to the root. Thus we gave each node on the distance i from the root the weight $1/c^i$, where c is a predefined constant. In our experiments we used the values $c = 2$. For comparison, we also ran tests with $c = 0.5$

Fig. 1. Performance test results of different strategies for choosing undefined literals



and $c = 1$. (Not that the Most-AND strategy is equivalent to the Weighted-AND strategy with $c = 1$.)

4 Performance analysis

Somewhat surprisingly it turned out that giving more weight to the AND nodes which are closer to the root node does not necessarily help. The weighting constant $c = 0.5$ gave also very good results and in some cases better than the Weighted-2-AND strategy.

We generated random sample attack trees with 5 leaves up to 29 leaves, at least 100 trees in each group, and measured the solving time with our optimized realization and with different strategies. The results are depicted in Fig. 1.

To estimate the complexity of our algorithms, we used the least-squares method to fit a function $a^{-1} \cdot b^n$ to the running times of our best strategy method. Since there is no reasonable analytical way to establish the time complexity of our algorithm, this approach provided a quick and

easy way to estimate it. The found parameters to fit the data points of the best solution (the (1)-AND method) optimally were $a = 109828$ and $b = 1.71018$. Hence we can conclude that the average complexity of our algorithm with our generated sample data is in the range of $\sim \mathcal{O}(1.71^n)$.

The average complexity estimations for all strategies were the following:

- Random strategy – $\mathcal{O}(1.90^n)$
- Weighted-2-AND strategy – $\mathcal{O}(1.78^n)$
- Weighted-1-AND strategy – $\mathcal{O}(1.71^n)$
- Weighted-0.5-AND strategy – $\mathcal{O}(1.75^n)$

However, it should be noted that differences between the Weighted-c-AND strategies were quite small and within the margin of error. Therefore, no conclusive results can be given regarding the different weighting constants. It is clear though that all the tested strategies are better than just choosing the leafs in random.

Currently the world's best #3-SAT problem solving algorithm by Konstantin Kutzkov ([16]) has the worst-case complexity $\mathcal{O}(1.6423^n)$. As #SAT problems can be parsinomically and in polynomial time converted to the #3-SAT problems (see [17], chapter 26), we can roughly compare our algorithm complexity and the #3-SAT problem solver complexity.

Direct comparison is however not possible for several reasons. First, our estimate is heuristic and is based on experimental data. Second, we are not only counting all the possible SAT solutions to the formula $\mathcal{F}$, but we actually have to generate many of them. At the same time, we are using optimizations described in Section 3.

Comparison with the result of Kutzkov still shows that our approach works roughly as well as one would expect based on the similarity of our problem setting to #SAT. It remains an open problem to develop more direct comparison methods and to find out whether some of the techniques of #SAT solvers could be adapted to the attack trees directly.

5 Approximation

Even though reimplementation in C++ and various optimization techniques described in Section 3 helped us to increase the performance of attack tree analysis significantly compared to [14], we are still practically limited to the trees having at most 30 leaves. Given the exponential nature of the problem, it is hard to expect substantial progress in the exact computations.

In order to find out, how well the exact outcome of the attack tree can be approximated, we implemented a genetic algorithm (GA) for the computations. (See [18] for an introduction into GAs.) Let us have an attack tree described by the Boolean formula $\mathcal{F}$ and the set of leaves $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ having the parameters as described in 3. We will specify the following concepts for our GA.

Individual: any subset $\sigma \subseteq \mathcal{X}$. The individuals are internally represented by bitstrings of length n, where 1 in position i means that $X_i \in \sigma$ and 0 in position i means that $X_i \notin \sigma$.

Live individual: $\sigma \subseteq \mathcal{X}$ such that $\mathcal{F}(\sigma := \mathsf{t}) = \mathsf{t}$, i.e. such that the value of $\mathcal{F}$ is t when all the literals of σ are set to t and all the others to f.

Dead individual: $\sigma \subseteq \mathcal{X}$ such that $\mathcal{F}(\sigma := \mathsf{t}) = \mathsf{f}$.

Generation: a set of p live individuals (where p is a system-wide parameter to be determined later).

Fitness function: $\mathsf{Outcome}_\sigma$. Note that σ must be alive in order for the fitness function to be well-defined.

Crossover: in order to cross two individuals σ_1 and σ_2, we iterate throughout all the elementary attacks X_i $(i = 1, \ldots, n)$ and decide by a fair coin toss, whether we should take the descendant's ith bit from σ_1 or σ_2.

Mutation: when an individual σ needs to be mutated, a biased coin is flipped for every leaf X_i $(i = 1, \ldots, n)$ and its status (included/excluded) in σ will be changed if the coin shows heads. Since σ is internally kept as a bit sequence, mutation is accomplished by simple bit flipping.

In order to start the GA, the first generation of p live individuals must be created. We generate them randomly, using the following recursive routine.

1. Consider the root node.
2. If the node we consider is a leaf, then include it to σ and stop.
3. If the node we consider is an AND-node, consider all its descendants recursively going back to Step 2.
4. If the node we consider is an OR-node, flip a fair coin for all of them to decide whether they should be considered or not. If none of the descendants was chosen, flip the coin again for all of them. Repeat until at least one descendant gets chosen. Continue with Step 2.

It is easy to see that the resulting σ is guaranteed to be live and that the routine stops with probability 1.

Having produced our first generation of individuals $\sigma_1, \ldots, \sigma_p$ (not all of them being necessarily distinct), we start the reproduction process.

1. All the individuals σ_i are crossed with everybody else, producing $\binom{p}{2}$ new individuals.
2. Each individual is mutated with probability 0.1 and for each one of them, the bias 0.1 is used.
3. The original individuals $\sigma_1, \ldots, \sigma_p$ are added to the candidate (multi)set. (Note that this guarantees the existence of p live individuals.)
4. All the candidates are checked for liveness (by evaluating $\mathcal{F}(\sigma = \mathtt{t})$) and only the live ones are left.
5. Finally, p fittest individuals are selected for the next generation.

The reproduction takes place for g rounds, where g is also a system-wide parameter yet to be determined.

Next we estimate the time complexity of our GA.

- Generating p live individuals takes $\mathcal{O}(np)$ steps.
- Creating a new candidate generation by crossing takes $\mathcal{O}(np^2)$ steps.
- Mutating the candidate generation takes $\mathcal{O}(np^2)$ steps.
- Verifying liveliness takes $\mathcal{O}(np^2)$ steps.
- Computing the outcomes of live individuals takes $\mathcal{O}(np^2)$ steps.
- Sorting out the p best individuals out of the $\binom{p}{2} + p$ individuals takes $\mathcal{O}(p^2 \log p))$ steps.

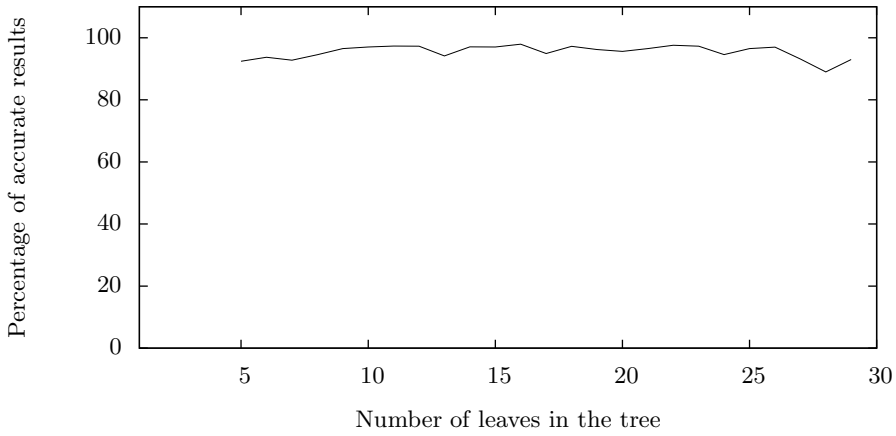
Since these steps are repeated for g generations, we can find the overall time complexity to be $\mathcal{O}(gp^2(\log p + n))$.

Of course, a GA does not guarantee that the final outcome is the best one globally. To find out, how large populations and how many iterations one has to use to hit the global maximum outcome with some degree of certainty, we performed series of tests.

First we generated a sample random set of about 6000 trees (having $n = 5 \ldots 29$ leaves) and computed the exact outcome for each one of them as described in Sections 3 and 4. Next we ran our GA for population sizes $p = 5, 10, 15, \ldots, 60$ and the number of iterations $1, 2, \ldots, 200$. We recorded the average running times and attacker's estimated outcomes, comparing the latter ones to the exact outcomes computed before.

As a result of our tests we can say that GA allows us to reach high level of confidence (say, with 90% accuracy) very fast. There are many possible reasonable choices for $p = p(n)$ and $g = g(n)$. For example, taking $p = 2n$ and $g = 2n$ allowed us to reach 89% level of confidence for all the tree sizes of up to 29 leaves (see Fig. 2). By accuracy we here

Fig. 2. Accuracy of genetics algorithm with $p = 2n$ and $g = 2n$



mean the ratio of the trees actually computed correctly by our GA when compared to the exact outcome.

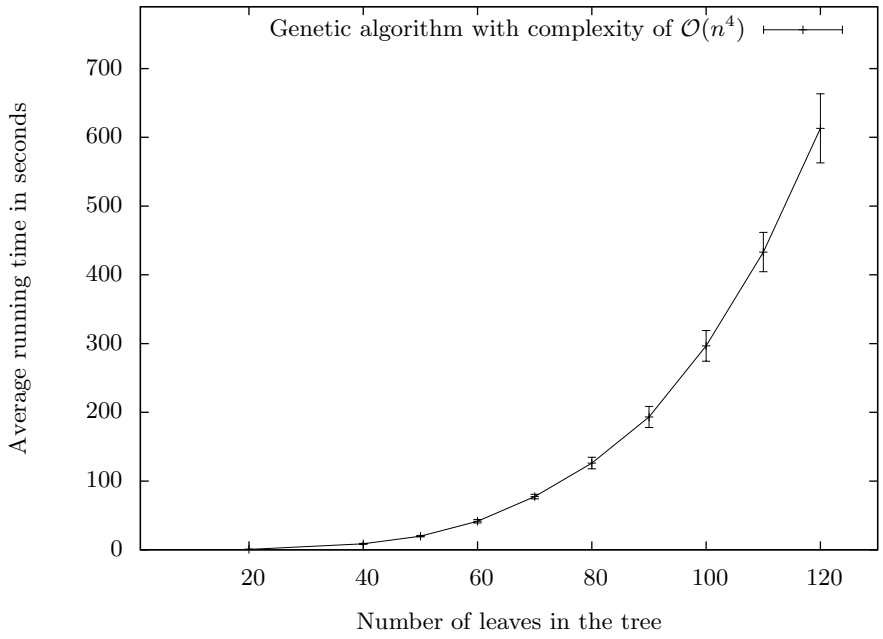
The theoretical time complexity of our GA is in this case $\mathcal{O}(n^4)$, which in reality required up to roughly 2 seconds for trees with less than 30 leaves on 2.33 GHz Intel Xeon processor. The same approach also enables us to process moderate size attack trees (70-80 leaves) in reasonable time (1-2 minutes). Attack trees of this size are helpful in analyzing real-life information systems and the multi-parameter attack trees can be now used in practical security analysis. The performance results for larger trees are given in Fig. 3. For each data point we generated 10 random trees and the average running times were measured for the genetic algorithm with parameters $p = 2n$ and $g = 2n$. The error bars represent the standard deviation of the average running time.

6 Conclusions and Further Work

In this paper we reviewed the method proposed by Jürgenson and Willemson for computing the exact outcome of a multi-parameter attack tree [14]. We proposed and implemented several optimizations and this allowed us to move the horizon of computability from the trees having 20 leaves (as in [14]) to the trees with roughly 30 leaves.

However, computing the exact outcome of an attack tree is an inherently exponential problem, hence mere optimizations on the implementa-

Fig. 3. Performance results of the genetic algorithm



tion level are rather limited. Thus we also considered an approximation technique based on genetic programming. This approach turned out to be very successful, allowing us to reach 89% of confidence within 2 seconds of computation for the trees having up to 29 leaves. The genetic algorithm is also very well scalable, making it practical to analyze even the trees having more than 100 leaves.

When running a genetic approximation algorithm, we are essentially computing a lower bound to the attacker's expected outcome. Still, an upper bound (showing that the attacker can not achieve *more* than some amount) would be much more interesting in practice. Hence, the problem of finding efficient upper bounds remains an interesting challenge for future research.

Another interesting direction is extending the model of attack tree computations. For example, Jürgenson and Willemson have also considered the serial model, where the attacker can make his decisions based on previous success or failure of elementary attacks [19]. It turns out that finding the best permutation of the elementary attacks may turn com-

puting the optimal expected outcome into a super-exponential problem, hence the use of good approximation methods becomes inevitable in that setting.

7 Acknowledgments

This research was supported by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and Estonian Science Foundation grant no 8124.

References

1. Vesely, W., Goldberg, F., Roberts, N., Haasl, D.: Fault Tree Handbook. US Government Printing Office (January 1981) Systems and Reliability Research, Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission.
2. Weiss, J.D.: A system security engineering process. In: Proceedings of the 14th National Computer Security Conference. (1991) 572–581
3. Schneier, B.: Attack trees: Modeling security threats. Dr. Dobb's Journal **24**(12) (December 1999) 21–29
4. Schneier, B.: Secrets & Lies. Digital Security in a Networked World. John Wiley & Sons (2000)
5. Convery, S., Cook, D., Franz, M.: An attack tree for the border gateway protocol. IETF Internet draft (Feb 2004) Available at http://www.ietf.org/proceedings/04aug/I-D/draft-ietf-rpsec-bgpattack-00.txt.
6. Byres, E., Franz, M., Miller, D.: The use of attack trees in assessing vulnerabilities in SCADA systems. In: International Infrastructure Survivability Workshop (IISW'04), IEEE, Lisbon, Portugal. (2004)
7. Buldas, A., Mägi, T.: Practical security analysis of e-voting systems. In Miyaji, A., Kikuchi, H., Rannenberg, K., eds.: Advances in Information and Computer Security, Second International Workshop on Security, IWSEC. Volume 4752 of LNCS., Springer (2007) 320–335
8. Edge, K.S.: A Framework for Analyzing and Mitigating the Vulnerabilities of Complex Systems via Attack and Protection Trees. PhD thesis, Air Force Institute of Technology, Ohio (2007)
9. Espedahlen, J.H.: Attack trees describing security in distributed internet-enabled metrology. Master's thesis, Department of Computer Science and Media Technology, Gjøvik University College (2007)
10. Moore, A.P., Ellison, R.J., Linger, R.C.: Attack modeling for information security and survivability. Technical Report CMU/SEI-2001-TN-001, Software Engineering Institute (2001)
11. Mauw, S., Oostdijk, M.: Foundations of attack trees. In Won, D., Kim, S., eds.: International Conference on Information Security and Cryptology – ICISC 2005. Volume 3935 of LNCS., Springer (2005) 186–198
12. Buldas, A., Laud, P., Priisalu, J., Saarepera, M., Willemson, J.: Rational Choice of Security Measures via Multi-Parameter Attack Trees. In: Critical Information Infrastructures Security. First International Workshop, CRITIS 2006. Volume 4347 of LNCS., Springer (2006) 235–248

13. Jürgenson, A., Willemson, J.: Processing multi-parameter attacktrees with estimated parameter values. In Miyaji, A., Kikuchi, H., Rannenberg, K., eds.: Advances in Information and Computer Security, Second International Workshop on Security, IWSEC. Volume 4752 of LNCS., Springer (2007) 308–319
14. Jürgenson, A., Willemson, J.: Computing exact outcomes of multi-parameter attack trees. In: On the Move to Meaningful Internet Systems: OTM 2008. Volume 5332 of LNCS., Springer (2008) 1036–1051
15. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM **5**(7) (1962) 394–397
16. Kutzkov, K.: New upper bound for the #3-sat problem. Inf. Process. Lett. **105**(1) (2007) 1–5
17. Kozen, D.: The design and analysis of algorithms. Springer (1992)
18. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
19. Jürgenson, A., Willemson, J.: Serial model for attack tree computations. In: Proceedings of ICISC 2009. (2009)