Effectiveness of Modular analysis on attack trees using Binary Decision Diagrams

VASIL PIRINSKI, University of Twente, The Netherlands

Abstract

Attack trees are a simple model used for system security that provides a systematic way to assess possible attacks on a system and quantify them using different metrics, such as total cost or damage caused. This is an NP-Complete problem. However, brute-forcing all possible attacks is too costly for large attack trees, so there are algorithms for making the analysis more efficient. One such algorithm transforms the attack tree into a Binary Decision Diagram (BDD). Solving this is exponential in time at worst, so we apply an existing method, called Modular analysis, where the work is split by first solving smaller components (modules). The goal of this research is to check the effect Modular analysis has on the performance of the BDD algorithm. We construct a corpus of attack trees of up to 260 nodes and analyze the computation times of our implementations for both methods. The results indicate that the time complexities of both methods are exponential, but Modular analysis tends to be significantly faster. Moreover, this increase in performance is dependent on the number of modules. However, this combined approach introduces some computational overhead, which causes negligibly longer computation times for attack trees that take fractions of a second to solve. The results we show are for calculating the minimal total cost.

Additional Key Words and Phrases: attack tree, Binary Decision Diagram, Modular analysis, quantitative analysis

1 INTRODUCTION

Attack trees(ATs) are graphical models used for representing the vulnerabilities of a system in the form of connected directed graphs. Their intuitive structure and power of expression make them widely used in the industry, from estimating the damage of DDoS attacks on computer systems [14], to analyzing the security of nuclear digital systems [17].

The so-called leaves of the tree are stand-alone independent attacks, also called Basic Attack Steps (BAS), that have some property associated with them, such as the cost of performing or the damage caused to the system. Moreover, they are connected through parent nodes that have certain activating conditions - AND/OR logical gates. At the very top, there is the root node, which represents the end goal of the attacker. Depending on what combination of the BAS-es are activated, the root could be reached, which we will refer to as a successful attack.

Fig. 1. Allowed types of nodes

In their simplest form attack trees are tree-structured, where each non-root node has exactly one parent. They can also be directed acyclic graphs (DAGs), where nodes could have multiple parents. Such is the case in Figure 2, where the node "Bribe employee" has two parents. We refer to such nodes as foster nodes, as suggested in [1]. In this research, we do not set a restriction on the number of parents, so we also refer to DAG-like structures as *attack trees* like in [12].

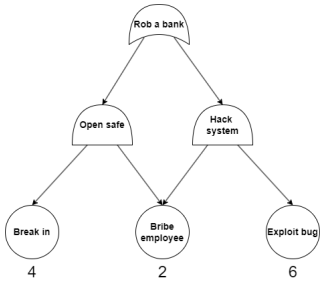

Fig. 2. simple DAG-like attack tree with costs

In this example, the attacker can open the safe by breaking in and bribing because it is an AND-gate. Similarly, bribing and exploiting a bug results in hacking the system. The final goal of robbing the bank is an OR-gate, so the attacker needs to either open the safe, hack the system, or do both.

Going back to the usage of ATs, to reach the root, a certain cost or indicator will be accumulated for activating the leaves. Of course, if all individual attacks are performed, the final goal will be reached, but we want a more optimal solution according to some metric - this is what we are looking for when quantitatively analyzing an AT. It is important because it shows where the biggest vulnerability of the system is and quantifies its security overall, e.g. minimal cost for breaking it. In this project, we only look at the minimal cost.

We can look at Figure 2 for a simple explanation of this metric. Each BAS ("Break in", "Bribe employee" and "Exploit bug") can either be performed or not, so there are eight possible combinations, out

TScIT 39, July 7, 2023, Enschede, The Netherlands

of which three are attacks (successful combinations): ["Break in", "Bribe employee"], ["Bribe employee", "Exploit bug"], ["Break in", "Bribe employee", "Exploit bug"], which have total costs 6, 8 and 12, respectively. Therefore, the minimal total cost is 6.

Doing this analysis efficiently is very important, since brute-forcing through all combinations would take exponential time. That is why algorithms for AT analysis have been developed, such as the simple bottom-up approach for tree-like ATs and the Binary Decision Diagram (BDD) algorithm which additionally works for DAG-like ATs [12]. The latter one consists of converting the attack tree into another type of DAG, namely a BDD, and then solving it. If we consider an attack tree to essentially be a boolean function, the BDD would be an encoding of that function. A simple case is shown in Figure 3.

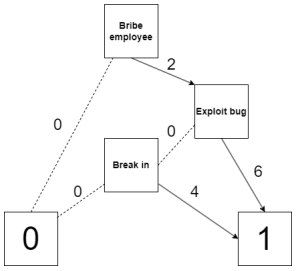

Fig. 3. BDD of the attack tree in Figure 2

In this example, we can see that each of the leaves of the attack tree is present as a variable in the BDD. We start with "Bribe employee". If it is not performed, we go directly to the end state 0, meaning that the goal was not accomplished. Looking at the attack tree representation, we see that both "Open safe" and "Hack system" require this node, so indeed if it is not activated, "Rob a bank" cannot happen. However, if "Bribe employee" is done, we go to the next variable "Exploit bug". We can either do it or choose not to and then do "Break in" to reach the final goal. Not doing both results in failure. One necessary clarification is that BDDs do not contain edge attributes like how we put the costs in Figure 3. We do this just to make it more understandable.

As for analyzing the BDD, we touched upon the fact that the BDD algorithm is for DAG-like ATs. That is because the bottom-up procedure used for tree-like ATs can produce incorrect results on such graphs [12]. However, a similar approach can be applied on the BDD representation instead. This method's effectiveness depends on the size of the generated BDD, which could theoretically be exponential in the number of nodes, but on average it tends to be relatively small [12]. An experimental evaluation of the algorithm has been conducted which seems to confirm this, but the conclusions are uncertain [1].

Furthermore, an idea has been suggested to apply this algorithm to smaller sections of the tree and use the intermediate results in the calculations of the whole [12]. This is done by first identifying the modules of an attack tree, then analyzing these parts separately, before finally solving the remaining component. A module is essentially a node that is the root of a sub-DAG that is only connected to the rest of the AT through that node. We refer to the application of this method to the BDD algorithm as Modular analysis. It makes sense for this approach to be an improvement for solving this exponential problem, because we are splitting it into sub-problems and therefore analyzing smaller BDDs with fewer variables, but it has not been confirmed in practice. Therefore, the goal of this project can be expressed with the following research question:

Question: *To what extent, if at all, does modular analysis improve the performance of the BDD algorithm for calculating the minimum cost of an attack tree?*

The steps taken for the completion of this project can be summarized in several steps. First, a sufficiently large corpus of realistic attack trees gets constructed. Second, the BDD algorithm and modular analysis are implemented and applied to the corpus. The results need to be structured and saved in a way that facilitates analysis of the data. Finally, appropriate visualization techniques are applied to compare the algorithms. Furthermore, we do simple correlation analysis to see how certain attributes affect the results.

We should mention that we use Python 3.10 for the whole project because it is a very intuitive general-purpose language that we already have plenty of experience with. It is especially appropriate for this project because of the wide range of libraries that are available. The most important ones we use are "networkx" - for representing and working with attack trees, while also giving us essentially complete freedom over the graph objects, "dd" - for constructing and using the BDDs, and "pandas", "matplotlib" and "seaborn" for data analysis.

2 RELATED WORK

As described in the previous section, the definitions of attack trees and the algorithms we are comparing are as discussed in [12]. However, there exists work on quantitative analysis for other types of models. In that same paper, a possible extension is mentioned in the form of dynamic ATs based on [2, 7]. This adds a new type of gate, namely the sequential AND, which requires the children nodes to be activated in a specific order. Similarly, another possible extension is to add defence gates and make an attack-defence tree [8]. These extensions allow for the representations of systems with more complex relations and in some cases findings about simple ATs can be generalized to apply to the more complex models too, such as computing the min time metric for dynamic ATs [13].

In some cases research for ATs can even be related to work on different models. For example, the use of the BDD algorithm with modularisation has been discussed in [15] for analyzing fault trees, which represent a system from the defensive side, although these two models are still quite similar.

Coming back to the AT model we are using, our work is related to research that experimentally evaluates the BDD algorithm in relation to different AT properties [1] based on the same literature as us. The authors found that the number of foster nodes is the most detrimental graph attribute for the performance of the algorithm, while the size of the AT seemingly has no impact, based on the Pearson coefficients for linear correlation. Our conclusions do not align with theirs, but this could be attributed to differences in the

implementation of the algorithm and the generation of the ATs used for the evaluation. The authors acknowledge that a more systematic approach can be used for generating the ATs and storing the results, to which we pay special attention.

Another relevant work is about attack time analysis via integer linear programming [13]. Although it is for dynamic ATs, it experimentally evaluates the proposed algorithm and compares its performance with a few other methods, including a variation of itself with Modular analysis. The results demonstrate the effectiveness of both integer linear programming and modular analysis, but not so much the combination of the two, possibly due to some hidden optimization in the used framework. This paper has been invaluable because of the proposed way of generating a corpus of ATs and the visualization techniques used for showing the results.

3 ATTACK TREES

In this section, we first give an extensive definition of attack trees. As mentioned before, we extend the definition to include DAG-structured ATs. Then, we will cover how an attack tree can be quantitatively assessed.

3.1 The attack tree model

An attack tree is a directed acyclic graph that models the vulnerabilities of a system that can be exploited by a malicious party. The final goal, represented by the root of the DAG, is further broken down into simpler steps. These can either be intermediate attacks or basic attack steps (BAS-es). The latter are the simplest actions that can be taken by the attacker and are the leaves of the tree-like structure. The former are lower-level attacks, which can be further decomposed into simpler ones. Furthermore, they are labeled with AND/OR logical gates that indicate their activation condition - how the activation of the children nodes determines the success of the intermediate attack.

Note that attack trees in our research, despite the name, are not necessarily tree-structured as a node can have multiple parents. We call such cases with more than one parent foster nodes. Lastly, we define an attack on the system as a subset of the basic attack steps that results in the activation of the root node. The ATs used in this research follow the formal definitions presented in some literature [12, 13]. A simple example is given in Figure 2. Now, we define the relevant terms similar to [12].

Definition 3.1. An attack tree is a tuple $T = (N, t, children)$ where:

- N: finite set of nodes;
- t: mapping of each node to the corresponding node $type$ with $type \in \{\text{BAS, OR, AND}\}$;
- $children$: mapping of each node to a list of its children;

Additionally, the following constraints need to be satisfied:

- (N, E) is a connected DAG, where
  $E = \{(parent, child) \in N^2 | child \in children(parent)\}$;
- $R_T \in T$ is a unique root such that
  $$\exists! R_T \in N. \forall v \in N. R_T \notin children(v);$$
- BAS_T are the leaves of T
  $$\forall v \in N. t(v) = \text{BAS} \Leftrightarrow children(v) = \varnothing;$$

Using this we can now define a successful attack:

- A is an attack such that $A \subseteq BAS_T$;
- f_T is an activation function for node v, such that:

$$f_T(v, A) = \begin{cases} 1 & \text{if } t(v) = \text{OR and } \exists u \in children(v). f_T(u, A) = 1, \\ 1 & \text{if } t(v) = \text{AND and } \forall u \in children(v). f_T(u, A) = 1, \\ 1 & \text{if } t(v) = \text{BAS and } v \in A, \\ 0 & \text{otherwise.} \end{cases}$$

- attack A reaches node v if
  $$f_T(v, A) = 1;$$
- A is a minimal attack on node v if
  $$\nexists A_1 \subseteq A. f_T(v, A_1) = 1;$$
- A is a successful attack if
  $$f_T(A) = f_T(R_T, A) = 1;$$
- $[[T]]$ is the set of all successful minimal attacks:

$$[[T]] = \{A \in BAS_T | (f_T(A) = 1) \ \& \ (\nexists A_1 \subseteq A. f_T(v, A_1) = 1)\};$$

3.2 AT metrics

Now that we understand what attack trees and successful attacks are, it is logical to consider some basis for the comparison of attacks. Many different metrics are used in practice such as the total cost of the attack or the damage caused to the system [12], but we will focus on the minimal cost.

- δ_T: mapping for the cost value of each BAS of T
- minimal total cost of T can be defined as:
  $$\min_{A \in [[T]]} \sum_{a \in A} \delta(a);$$

The last definition simply expresses the minimal total cost as the minimum sum of the costs of the BAS-es that are part of each successful minimal attack. However, to find and compare all possible attacks would be exponential in time, so there is a need for an algorithm that can do this more efficiently.

4 BDD ALGORITHM

It has been shown that tree-like ATs can be analyzed very efficiently with a simple bottom-up algorithm, but this method can produce faulty results for DAG-like ones [13]. Therefore, an alternative approach has been proposed, where the AT is turned into a Binary Decision Diagram, which is then solved in a similar bottom-up fashion.

A binary decision diagram is a directed DAG-like model used for representing boolean functions. It allows for efficiently and intuitively storing the relations between variables. Moreover, an attack tree can be turned into a BDD, the size of which is at worst linear in the number of leaves but tends to be of manageable size on average. This largely depends on the variable ordering when constructing the BDD, but finding such ordering is NP-hard [4]. There are some methods for optimizing this, but we have chosen to take the variables in order. More specifically, we traverse the attack tree using breadth-first search to represent it as a boolean expression. Then, we use the Python library 'dd' to create a BDD object from that expression.

For example, the AT from Figure 2 can be expressed as "((Break in) AND (Bribe employee)) OR ((Bribe employee) AND (Exploit bug))" with a simple recursive bottom-up approach. This boolean expression, then, is given as input to 'dd', which produces the BDD from Figure 3.

To analyze the BDD, we use a recursive method based on what is proposed in [13]. It is a bottom-up algorithm that works for semiring attribute domains, meaning that it can be used for calculating different metrics with specific properties. However, for this study, we will solely consider the metric of minimal total cost, since we are only interested in the calculation time. The pseudo-code for the implementation used in this study is given in Algorithm 1.

Algorithm 1: get_min_cost

Input: *BDD*, *node*, *costs*
Output: minimum total cost of *BDD* starting from *node*

```
if node → value = 1 then
    return 0;
else if node → value = −1 then
    return ∞;
else
    return min(
    get_min_cost(BDD, node → low, costs),
    get_min_cost(BDD, node → high, costs) + (costs → node)
    );
```

A small clarification about the algorithm is in order because it is fitted to the 'dd' library. First, we should mention that the BDD object does not contain the costs of the attacks, but they are instead stored in a separate dictionary, which you can see is passed as an argument in the function. This is because BDDs simply represent a boolean expression, so the variables do not have attributes present in the attack tree like the cost. Second, each node of the BDD has some attributes that we use for traversing the graph. The *high* and *low* attributes are pointers to the nodes that follow if the current node is True or False, respectively.

It should become clearer with an example. We use the BDD from Figure 3 again. We will substitute "get_min_cost(BDD, node, costs)" with "solve(node)" for the sake of readability.

$$solve(\text{"Bribe employee"}) = min(solve(\text{"-1"}), solve(\text{"Exploit bug"}))+2)$$
$$solve(\text{"-1"}) = \infty$$
$$solve(\text{"Exploit bug"}) = min(solve(\text{"Break in"}), solve(\text{"1"})) + 6)$$
$$solve(\text{"Break in"}) = min(solve(\text{"-1"}), solve(\text{"1"})) + 4)$$
$$solve(\text{"1"}) = 0$$
$$solve(\text{"Break in"}) = min(\infty, 0 + 4) = 4$$
$$solve(\text{"Exploit bug"}) = min(4, 0 + 6) = 4$$
$$solve(\text{"Bribe employee"}) = min(\infty, 4 + 2) = 6$$

5 MODULAR ANALYSIS

In the previous section, we explained the details relevant to the algorithm we aim to improve. Now, we will go over the concept of modules and how we apply them. We follow the formal definitions as written in [13], but essentially a module in an attack tree is a node, for which all nodes reachable from it cannot be reached from the root through any path not including that node. Therefore, it forms a sub-DAG that is connected to the attack tree at a single point. So, the idea of modular analysis is to replace that sub-DAG with a single basic attack step(a leaf node), which has as value the result of solving it, e.g. the total minimum cost of the module.

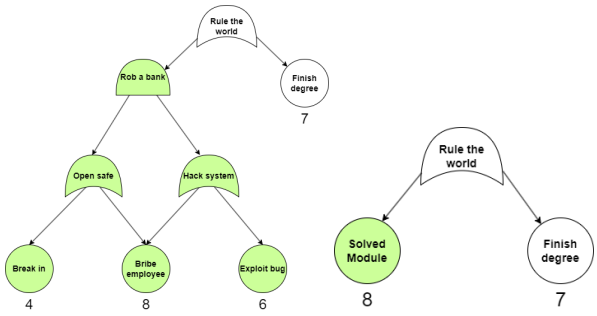

Fig. 4. AT with a module in green

Fig. 5. AT with solved module

The nodes marked in green in Figure 5 are part of the sub-DAG formed by the module "Rob a bank". We already solved this AT from Figure **??**. Therefore, we can just replace the entire sub-DAG with a new BAS node that has its solution as a cost, which is what we see in Figure 6.

For our implementation, we have a component that determines the modules of a given attack tree based on [5]. To do that we traverse the attack tree two times using depth-first search, once to mark the nodes with the visiting order and a second time to discover the modules based on the previously assigned numbers. The next step is, for each module, to solve the sub-DAG and construct a simplified version of the attack tree. We do this in such an order, that each

one is checked after all modules that are part of its sub-DAG. Then, once we are left with an attack tree without modules, we solve it using just the BDD algorithm.

One relevant detail about this approach is that our implementation does not use multi-threading. On the one hand, this makes the comparison of the algorithm and the modified version more fair because the same amount of resources are used, and on the other hand, this simplifies some of the components to be implemented. Most importantly, we do not need to worry about concurrently changing an attack tree for multiple modules and introducing more overhead by complicating the order of execution further. However, we consider the possibility of multi-threading to be quite important, so it will be discussed in Section 7.2.2.

6 EXPERIMENTS

6.1 Attack tree generation

One very important aspect of the experimentation is the data we conduct it on. For meaningful results, we want it to be of sufficient size, consist of attack trees from the industry, and also be diverse enough to capture some relations between different attack tree properties and the results. Using graphs from the industry such as an attack tree of existing big systems would be ideal, but making these publicly available would pose a security risk, so this is not an option for us. On the other hand, the ones found in academic literature are quite small. Therefore, we construct our own attack trees using an approach similar to what is done in [12], where large attack trees are generated by combining smaller ones.

We use nine attack trees of sizes between 8 and 25 nodes as building blocks, listed in Table 1. Some of them are for dynamic trees or attack-defence trees, in which cases we replaced the sequential AND-gates with normal AND-gates and removed the defence components, respectively. We consider these ATs to be appropriate examples of what can be seen in reality and they vary in characteristics.

Source	|N|
[3] Fig. 3	8
[3] Fig. 5	21
[3] Fig. 7	25
[10] Fig. 1	12
[10] Fig. 8	20
[10] Fig. 9	13
[2] Fig. 1	16
[6] Fig. 2	19
[9] Fig. 1	15

Table 1. Sources for the ATs we use as building blocks

Those get combined or mutated in three ways: we combine the current tree with a few blocks by adding a common root, we replace a random leaf of the attack tree with a block, which is essentially just adding a module, or we take a random number of leaves and merge them. A visual representation of these three operations is given in Figure 6.

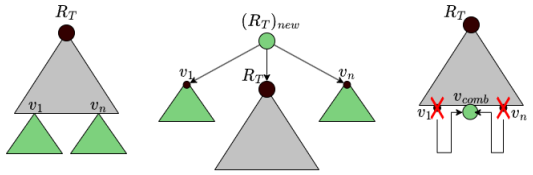

Fig. 6. Operations for generating ATs: 1) add blocks as sub-DAGs. 2) make a new root node R_{Tnew} and add our AT and add sub-DAGs. 3) merge BAS-es together

This continues until the graph is within the desired range, e.g. between 40 and 50 nodes, and the process is reset if it exceeds the upper limit. This is not very efficient but the generation of the corpus is done only once and it takes seconds, so it is sufficient. By using this method, we generate a corpus of 2300 attack trees of sizes between 20 and 250 nodes (100 ATs for each range of 10 nodes).

6.2 Approach

Our initial method of comparison was to simply run both algorithms on each graph and note the time. However, we discovered that when the analysis of an attack tree happened almost instantaneously, which was the case for most of the ATs with less than 100 nodes, the times were rounded to either zero or a very small value, which ruined the visualization and analysis of the produced data.

Instead, we opted for setting a boundary for each benchmark, where each attack tree is continuously ran through each analysis method for at least 5 seconds and at most 1000 seconds. After every successful solving of the AT, we check if we are past the lower limit, and if so, we divide the time that the benchmark has been running by the number of executions and end the process. And if the first analysis does not finish within the upper limit, we write 1000 as the time it took and continue.

Besides the benchmarks, for each attack tree, we also calculate some graph properties. Currently, these are the number of each type of node, as well as the minimum, mean, and maximum values for the depth (distance from the root to leaves), in-degree and out-degree. We also calculate the root-to-foster and foster-to-leaf distances as described in [1].

Furthermore, we analyze the relationship between these properties and the time it took for both algorithms to finish using the Pearson correlation coefficient. For the interpretation of the absolute values, we use the ranges from Table 2 as suggested in [16]. As for the sign of the value, for the times of the algorithms, a positive value indicates that increasing this property results in an increase in the time, whereas for the ratio of the times, a positive value means that this property improves how much better Modular analysis is.

r-value	Correlation
0.00-0.09	Negligible
0.10-0.39	Weak
0.40-0.69	Moderate
0.70-0.89	Strong
0.90-1.00	Very Strong

Table 2. Interpretation of Pearson correlation coefficient (absolute values)

We calculate these coefficients using the '*corr*' function provided in the Python library '*pandas*'. Moreover, this is done once with the timing values and once with their log to check for both linear and exponential relationships. Another important detail is that we only consider the data points where the two algorithms managed to finish the analysis within the limit of 1000 seconds, so that the potential correlation we find is not affected by it.

6.3 Results

After performing the steps described above, we end up with a table of all attack trees, their timings for the BDD algorithm and the modular analysis, and the graph properties described above. This allows us to experiment with various visualization techniques without again having to run any of the code described earlier. In this subsection, we show the representations we found appropriate and provide an explanation and interpretation for them.

6.3.1 *Overview of performance.*

We think it is suitable to start with a simple log-log scatter plot as can be seen on Figure 7. Each attack tree is plotted according to the time it took with the two approaches. Also, the diagonal line is plotted. Naturally, if the BDD algorithm is faster for a specific attack tree, we would get a point over the line. Conversely, if Modular analysis makes the process faster, the point would be under the line.

A scatter plot is definitely suitable because of the substantial number of instances we have. Additionally, we made it logarithmic because our data spans a wide range of values, so the points, where the analysis took a fraction of a second, become indistinguishable because of the cases that take 1000 seconds.

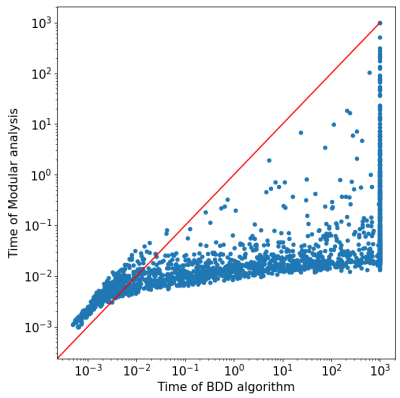

Fig. 7. Comparison of the times of the two algorithms on the log scale

Looking at Figure 7, we observe a clear trend. While the time it takes for the BDD algorithm continues to increase, most of the values for Modular analysis roughly form a line with a very small incline. Furthermore, there are two interesting details on the left and right sides of the plot. First, exactly at the 1000 seconds mark for the BDD algorithm, we notice a vertical line. This is because the calculation has an upper cap of 1000 seconds, meaning that the BDD algorithm was not able to finish, whereas with modular analysis the calculation of the minimal total cost was successful. Second, on the left side until around the 0.01 mark, we notice a lot of attack trees, for which the BDD algorithm performed better. We believe this is a result of the overhead of looking for modules and splitting the problem into smaller instances, where the initial attack tree can already be solved almost instantaneously. However, this overhead is clearly compensated for in instances that take longer to solve with the BDD algorithm.

These results seem conclusive but there are a lot of instances closer to the diagonal line. We found that the relative number of modules in the attack tree can help to understand this. More precisely, we take the ratio of the number of modules to the number of nodes with a logical gate (intermediate nodes and root). You can see in Figure 8, this value is shown as the color of each dot according to the scale on the right of the plot.

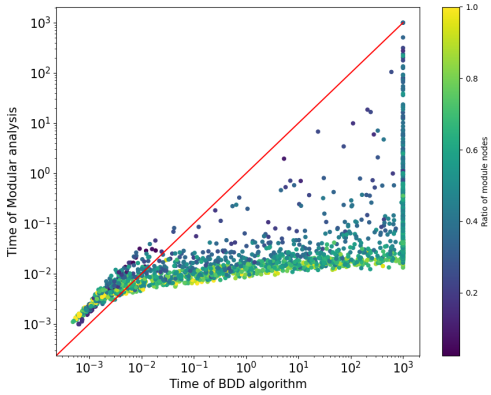

Fig. 8. Comparison of the times of the two algorithms on the log scale with the module ratio of each data point

Now, it is easier to distinguish the overlapping points at the bottom-left corner. Furthermore, we can observe that the data points are roughly grouped according to the colors. So, the method with Modular analysis appears to perform better the more modules there are. As for the cases on the right side of the plot that are closer to the red line, ignoring the ones at the upper limit of 1000 seconds, we can see that they have very low values for the ratio of modules. However, these cases do not seem to be that frequent.

In Figure 9, we see for each range of graph sizes, how the module ratio is distributed. For a proper comparison of the two algorithms we would want something closer to a uniform distribution, but that is not the case because of our generation method. After all, most building blocks we use have multiple modules, so the bigger the attack trees we create, the less likely it is to have an extremely

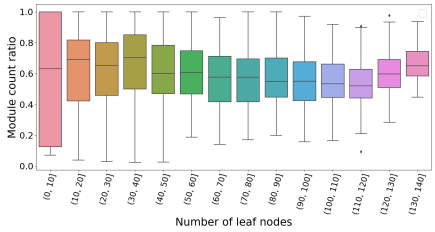

Fig. 9. Distribution of relative module count for each range of leaf nodes

low value. Additionally, the modification operation, which merges leaves of the attack tree together, makes it less likely to see high values. Therefore, we see the boxplots becoming smaller as the graphs become larger.

This means that it is harder to reliably quantify the performance increase that Modular analysis provides as a whole. Despite that, we can still make the conclusion that the higher the relative count of the modules is, the bigger the positive impact. And, of course, modular analysis would be slightly slower if the attack tree has no modules, since it is doing the work of checking for modules and then just solving the attack tree with the BDD algorithm.

6.3.2 Performance comparison relative to the AT size.
In this part, we demonstrate the difference in performance between the BDD algorithm and Modular analysis relative to the size of the attack tree, by grouping the data in ranges based on the number of leaf nodes. We provide a line chart of the moving median for an overview and a plot with boxplots of each grouping for more detailed insight. Also, both plots use the logarithmic scale for the solving time, so that we can compare their relative behavior for smaller values.

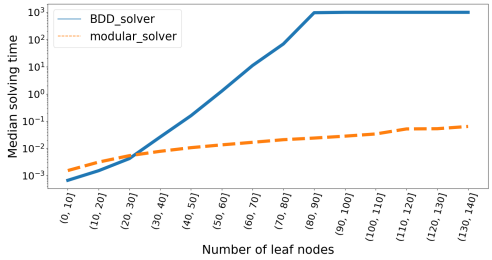

Fig. 10. Comparison of the median times of the two algorithms on the log scale as the number of leaves increases. Calculation for each AT is interrupted at 10^3 seconds

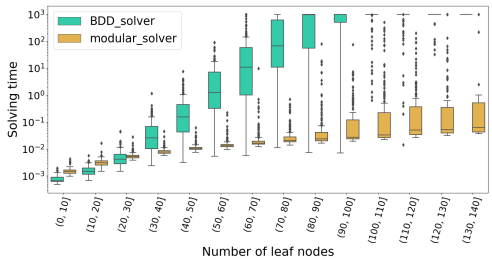

Fig. 11. Comparison of the distributions of the times of the two algorithms on the log scale as the number of leaves increases. Calculation for each AT is interrupted at 10^3 seconds

We already showed that, for attack trees that take fractions of a second to solve, the BDD algorithm tends to be faster. From Figures 10 and 11, we can conclude that this is generally the case for attack trees with less than 30 leaves. We consider this to be insignificant since these algorithms are meant to be used for substantially bigger graphs where the brute-force approach is computationally unfeasible.

It is also apparent that the lines for each algorithm appear to be straight (ignoring that the BDD algorithm hit the upper limit of 1000 seconds). This tells us that both are exponential in the number of leaves, but it is clear that the line of Modular analysis has a much lower incline, approximately four times. This makes sense since it essentially does a bit of extra work to make an attack tree smaller before applying the BDD algorithm, so this size reduction has a substantial impact. However, it is worth noting that there are a lot of outliers, so there are likely other important factors that affect the computation time besides the number of leaves.

6.3.3 Analysis of the effect of graph properties.
Now, we show the results of the analysis of the graph properties using a heatmap containing the Pearson correlation coefficient between each pair of attributes in our dataset. Rows 1 and 2 contain the coefficients for each property for the time of the BDD algorithm and Modular analysis, respectively. The third row is for the ratio of the time it took the latter compared to the former, meaning that the ratio is how many times faster the second algorithm is. The columns are some of the attributes mentioned in Section 6.2. Some less meaningful or repetitive ones were omitted for readability.

Fig. 12. Pearson correlation coefficient between the computation times of the algorithms and the graph attributes

The heatmap on Figure 12 shows a weak to moderate correlation between the size of the attack tree, e.g. number of nodes, gates, and leaves, and the computation time for the BDD algorithm. However, this is not the case with Modular analysis, where it is between negligible and weak.

As for the ratio showing how much faster the second approach is, there is a moderate linear correlation with the number of modules in an attack tree. Additionally, we can see that the size of the graph and the mean root-to-foster distance have a weak correlation with the ratio. However, if we take the logarithm of the time of the two algorithms, we can see a lot more conclusive numbers.

Fig. 13. Pearson correlation coefficient between the logarithm of the computation times of the algorithms and the graph attributes

For Figure 13, since we are taking the logarithm of the time, seeing a linear correlation, would indicate an exponential relationship with the original value. And indeed, both algorithms show a strong exponential correlation between the size of the graph and the time it takes to do the analysis. We also see that the depth and foster-to-leaves values seem to impact the performance of the second algorithm more than the first one.

One important observation is that there is a weak negative exponential correlation between the percentage of nodes that are modules and the performance of the algorithm with Modular analysis. And if we only look at larger attack trees (more than 30 leaves), this correlation is moderate with an r-value of -0.47. This is consistent with our findings from Figure 8 where the y-axis is with a logarithmic scale and we can see a roughly horizontal grouping of the colors.

7 CONCLUSION

7.1 Discussion

In this paper, we compared the performance of our implementation for the BDD algorithm with and without Modular analysis. We found that Modular analysis introduces some overhead to the BDD algorithm, which makes it slightly slower for smaller DAGs, but it helps tremendously when analyzing large ones.

We also performed simple correlation analysis between the graph properties and the computation times of the two algorithms, which supports our observations that both algorithms appear to be exponential in the size of the graph and that the impact of Modular analysis is heavily dependant on the number of modules. So, at worst it is still exponential in the number of BAS-es in cases with an insignificant number of modules.

However, we used the Pearson correlation coefficient, which is sensitive to outliers, so a more extensive look could be taken at the distribution of our data and potentially other statistical tools such as Spearman's correlation coefficient.

One setback caused by the time constraints of this project is that we had to set an upper limit of 1000 seconds for the run-time of the algorithms for each AT. It would be interesting to see how our results would look like if we could remove that limit and got the complete data.

7.2 Future work

Overall, this research has accomplished the goals agreed upon at the beginning. Throughout the process, we came up with several ideas for how the project could be expanded, but due to time constraints, those could not be realized. We will cover them in this section.

7.2.1 *Corpus of ATs.*

The findings of this research rely on the quality of the dataset they were made on. At the moment of writing, the only issue we are aware of is that because of the way we generate attack trees, the distribution of graph attributes is uncertain. This limits our ability to confidently make general statements about the effect of specific properties on the performance of either algorithm, but this was either way not the main point of our research.

However, the methods of evaluation we performed can be directly applied to any dataset. Therefore, if someone produces a good-quality corpus of attack trees by a better method of generation or by gathering them from real systems, our work can easily be applied to it and a comparison of the results can be made. This is important because there is no benchmark for these methods on real-life ATs, but perhaps an agreement can be made with companies for them to use our work to evaluate their systems and anonymously provide benchmark data without sharing the ATs and risk exposing their vulnerabilities.

7.2.2 *Multi-threading modular analysis.*

As described in Section 5, in our implementation we do not use multi-threading to solve modules in parallel. This means that our comparison is based on the number of computations, rather than efficiency in practice. So, one direction this project could go is to implement this variation of the Modular analysis method and compare it with our results. After all, the motivation for this research is to be able to analyze complex systems faster, which this will probably accomplish.

Even with sequential handling of the modules, we have observed that we need to be careful with the order since some modules might be part of the sub-DAGs of others, and therefore need to be executed first. For parallel processing, these same relations can be used to make sure that only modules that do not depend on any other modules are being analyzed. Therefore, such a continuation of our work should be easy to perform.

7.2.3 *BDD variable ordering.*

In Section 4 we touched upon the fact that the ordering of the variables is important for the size of the BDD. Since both the BDD algorithm and the one with modular analysis depend on this, finding a consistent way to get the BDD is logical if one aims to optimize our implementation.

One idea that we thought of but did not try because of time constraints is to avoid the problem by decreasing the time limit and repeating the BDD transformation step multiple times. So, whenever the limit is reached, we shuffle the boolean formula to one we have not yet tried to generate a BDD with and attempt to solve this new one.

Another direction is to try to implement optimization techniques for better variable ordering [11, 18]. In any case, it would be very interesting to see how improvements in the implementation of the BDD algorithm could change the results.

REFERENCES

[1] A. Afia. 2022. The impact of graph properties on the complexity of attack tree analysis. http://essay.utwente.nl/92052/

[2] Florian Arnold, Dennis Guck, Rajesh Kumar, and Mariële Stoelinga. 2015. Sequential and Parallel Attack Tree Modelling. In *Computer Safety, Reliability, and Security*, Floor Koornneef and Coen van Gulijk (Eds.). Springer International Publishing, Cham, 291–299.

[3] Florian Arnold, Holger Hermanns, Reza Pulungan, and Mariëlle Stoelinga. 2014. Time-Dependent Analysis of Attacks. In *Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 285–305.

[4] B. Bollig and I. Wegener. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comput.* 45, 9 (1996), 993–1002. https://doi.org/10.1109/12.537122

[5] Y. Dutuit and A. Rauzy. 1996. A linear-time algorithm to find modules of fault trees. *IEEE Transactions on Reliability* 45, 3 (1996), 422–425. https://doi.org/10.1109/24.537011

[6] Marlon Fraile, Margaret Ford, Olga Gadyatskaya, Rajesh Kumar, Mariëlle Stoelinga, and Rolando Trujillo-Rasua. 2016. Using Attack-Defense Trees to Analyze Threats and Countermeasures in an ATM: A Case Study, Vol. 267. https://doi.org/10.1007/978-3-319-48393-1_24

[7] Ravi Jhawar, Barbara Kordy, Sjouke Mauw, Sasa Radomirovic, and Rolando Trujillo-Rasua. 2015. Attack Trees with Sequential Conjunction. *CoRR* abs/1503.02261 (2015). arXiv:1503.02261 http://arxiv.org/abs/1503.02261

[8] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. 2011. Foundations of Attack–Defense Trees. In *Formal Aspects of Security and Trust*, Pierpaolo Degano, Sandro Etalle, and Joshua Guttman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–95.

[9] Barbara Kordy and Wojciech Wideł. 2018. On Quantitative Analysis of Attack–Defense Trees with Repeated Labels. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 325–346.

[10] Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. 2015. Quantitative Attack Tree Analysis via Priced Timed Automata. In *Formal Modeling and Analysis of Timed Systems*, Sriram Sankaranarayanan and Enrico Vicario (Eds.). Springer International Publishing, Cham, 156–171.

[11] Minh Lê, Josef Weidendorfer, and Max Walter. 2014. A Novel Variable Ordering Heuristic for BDD-based K-Terminal Reliability. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 527–537. https://doi.org/10.1109/DSN.2014.55

[12] Milan Lopuhaa-Zwakenberg, Carlos E. Budde, and Marielle Stoelinga. 2022. Efficient and Generic Algorithms for Quantitative Attack Tree Analysis. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–18. https://doi.org/10.1109/tdsc.2022.3215752

[13] Milan Lopuhaä-Zwakenberg and Mariëlle Stoelinga. 2023. Attack time analysis in dynamic attack trees via integer linear programming. arXiv:2111.05114 [cs.CR]

[14] Ronierison Maciel, Jean Araujo, Jamilson Dantas, Carlos Melo, Erico Guedes, and Paulo Maciel. 2018. Impact of a DDoS attack on computer systems: An approach based on an attack tree model. In *2018 Annual IEEE International Systems Conference (SysCon)*. 1–8. https://doi.org/10.1109/SYSCON.2018.8369611

[15] Karen A. Reay and John D. Andrews. 2002. A fault tree analysis strategy using binary decision diagrams. *Reliability Engineering System Safety* 78, 1 (2002), 45–56. https://doi.org/10.1016/S0951-8320(02)00107-2

[16] Patrick Schober, Christa Boer, and Lothar A. Schwarte. 2018. Correlation coefficients. *Anesthesia amp;amp; Analgesia* 126, 5 (2018), 1763–1768. https://doi.org/10.1213/ane.0000000000002864

[17] Jae-Gu Song, Jung-Woon Lee, Cheol-Kwon Lee, Kee-Choon Kwon, and Dong-Young Lee. 2012. A cyber security risk assessment for the design of IC systems in nuclear power plants. *Nuclear Engineering and Technology* 44 (12 2012). https://doi.org/10.5516/NET.04.2011.065

[18] Leslie G. Valiant. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.* 8, 3 (1979), 410–421. https://doi.org/10.1137/0208032 arXiv:https://doi.org/10.1137/0208032