The impact of graph properties on the complexity of attack tree analysis

ALY AFIA, University of Twente, The Netherlands

Attack Trees are tree-like diagrams that represent the logical steps by which a target may be attacked. They may be analysed to extract meaningful information about attacks on the given system. As the target systems become ever more complex, so does its Attack Tree, as more information is incorporated into them. This paper studies which graph metrics—such as the number of nodes, or the depth of the tree—affect the complexity of Attack Tree analyses and how they are affected. It appears that the most effective metric that was explored is the number of so-called foster nodes in an Attack Tree.

Additional Key Words and Phrases: Attack Trees, Graph Metrics, Binary Decision Diagram

1 INTRODUCTION

As information systems become increasingly complex, the utility and application of Attack Trees have comparably increased [18]. Attack Trees, Binary Decision Diagrams, and Quantitative Attack Tree Analysis have been used extensively throughout this research.

Attack Trees have been prominent in the cybersecurity field for many years, following from the fact that they were introduced in the late 1990s [17]. Most importantly, Attack Tree analysis is widely used even in recent years [13]. Recently, a new method to calculate Attack Tree metrics based on so-called Binary Decision Diagrams (see Sections 3.1 and 3.2) was introduced [6]. While finding the BDD corresponding to the AT is NP-hard in the worst case, Budde & Stoelinga argue that this calculation is quite fast in practice in most cases.

To that end, we are interested in what factors affect the time complexity of Attack Tree analysis, precisely which graphical factors of ATs. In particular, there is a focus on so-called "foster" nodes (see Section 3.1.1) because it was shown in [6] that analysis on tree-like ATs is considerably faster than on DAG-structured ATs.

2 PROBLEM STATEMENT

Even though there has been research done to find efficient algorithms for computing security metrics of the different types of Attack Trees, there has not been any research on how the computational speed of those algorithms is affected by the specific AT on which they are being invoked. Thus, the primary focus of this paper is not on the tree analysis algorithms themselves but on how their computation times are affected by different properties of the Attack Trees they are analysing.

2.1 Research Question

The problem statement then leads to the following main research question: *How does the computation time of Attack Tree metrics depend on the structural parameters of the Attack Tree?* We refine this into the following sub-questions, which are defined by the different structural parameters that will be considered:

(1) How does the size (number of nodes) of the AT affect the computation time of its metrics?
(2) How does the median of the out-degree (number of children) of the nodes in the AT affect the computation time of its metrics?
(3) How does the mean in-degree (number of parents) of the nodes in the AT affect the computation time of its metrics?
(4) How does the depth of the AT affect the computation time of its metrics?
(5) How does the median of the distance between the root and foster nodes in the AT affect the computation time of its metrics?
(6) How does the median of the distance between the foster nodes and the leaves in the AT affect the computation time of its metrics?
(7) How does the number of foster nodes in the AT affect the computation time of its metrics?

3 PRELIMINARIES AND RELATED WORK

3.1 Attack Trees

Attack Trees (ATs) provide a representation of all possible attacks on a system using a tree-like structure. They are structurally defined as singe-rooted Directed Acyclic Graphs (DAGs). An AT is composed of three types of nodes:

(1) *Root node*: The top node in the tree, which represents the target of the attack.
(2) *Basic Attack Step (BAS)*: The leaf nodes in the tree, which represents the initial conditions or steps that need to be fulfilled.
(3) *Intermediate Event*: The middle nodes, or Intermediate Attack Steps, which have children that may be either intermediate events themselves, or BASs. Each intermediate event is labelled with a logical gate (either *AND* or *OR*). The gate represents how its children need to combine for it to be achieved/activated.

In any given AT, all nodes represent the immediate consequences for their children, while BASs represent actions that the attacker may take. When such a step is taken, the BAS representing that step is considered activated. When an Intermediate Event's children are activated in the combination dictated by its logical gate, then that Event is considered activated. An attack is considered successful when the root node has been activated. The leaf nodes represent the first steps that the attacker can take.

It is possible to extend Attack Trees in order to embed more information about the system. For example, the most notable extension to an AT could be constricting a node's activation condition by imposing a certain order in which its children should be activated. This changes a Static Attack Tree (SAT) into a dynamic one. Dynamic Attack Trees will not be discussed in this paper. We refer the interested reader to [5].

TScIT 37, July 8, 2022, Enschede, The Netherlands

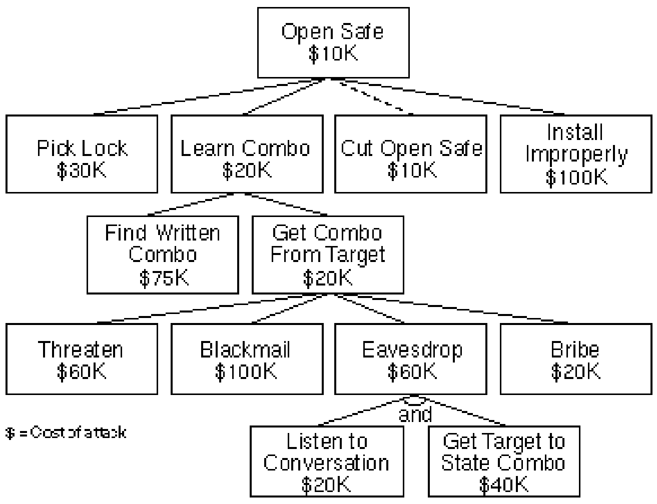


Fig. 1. An example Attack Tree

Finally, another extension — which will be used extensively in this paper — is to attach parameters to each BAS. Those parameters are chosen based on the specific security metric being calculated. For example, consider that the security metric that should be calculated is the quickest attack. Then, the relevant parameter that each node could have is the time estimated for its completion.

3.1.1 Foster Nodes. As explained above, an AT is defined as a DAG. This means that any non-root node could have more than one parent. Since such nodes will be explored extensively in this paper, they will be referred to as foster nodes.

One could refer to Figure 1 for an example Attack Tree, taken from [15]. The root note represents the goal of the attacker; in this case, to open a safe. As seen in the Event node *Eavesdrop*, it has a logical gate *AND*. Any other nodes have *OR* gates. Furthermore, one can see that each node has a cost value attached to it. While each BAS node has a cost value attached as expected, Intermediate Event nodes also have a value attached. According to [15], the values propagate upwards, such that the cost value in the root node is the cost of the cheapest attack.

Event nodes with *OR* gates have the value of their cheapest child, while *AND* gates have the value of the sum of their children. For example, the node *Eavesdrop* has a value of \$60K because the sum of its two children is $\$20K + \$40K = \$60K$. On the other hand, consider the node *Get Combo From Target*, which has a value of \$20K. This is because \$20K is the cheapest of this node's children; $min(\$60K, \$100K, \$60K, \$20K) = \$20K$. We may discard the dotted line between the root node and the node *Cut Open Safe*.

3.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) offer a very compact and intuitive way to represent Boolean functions through a tree structure. Since Attack Trees represent a particular Boolean function (the attack has either succeeded or not), we can exploit the speed and efficiency of the tree structure of BDDs by converting an AT into a BDD [6]. This

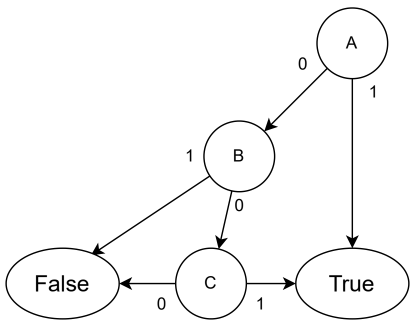


Fig. 2. An example Binary Decision Diagram

is especially useful when dealing with DAG-structured ATs. The size of the BDD can vary greatly, from linear to exponential, in the number of variables. It also depends on the order of the variables. Unfortunately, the problem of finding the optimal variable ordering is NP-hard (but can be improved, as discussed in [4]). However, the BDD encodings of ATs tend to be small [6].

Consider Figure 2, which shows an example BDD representing the Boolean formula $f = A \lor (\neg B \land C)$.

3.3 Quantitative Attack Tree Analysis

An array of algorithms have been researched for analysing different Attack Tree structures. It is essential to understand the algorithms employed in each AT structure, as they may affect the results of this paper. Appreciating the algorithm that would be employed given a specific AT will help determine which structural aspects of that AT affect the time for the algorithm to complete. Table 1 provides an overview of the different algorithms to compute a general class of metrics.

Table 1. Taken from [6]

Metric	Static tree	Dynamic tree	Static DAG	Dynamic DAG
min cost	BU [14, 15, 16]	BU [4]	MTBDD [17] $\mathcal{C}$-BU [18]	PTA [8]
min time	BU [14, 19]	APH [9] BU [4]	Petri nets [12]	PTA [8]
min skill	BU [14, 20]	BU [4]	$\mathcal{C}$-BU [18]	—
max damage	BU [14, 19, 20]	BU [4]	MTBDD [17] DPLL [7]	PTA [8]
probability	BU [6, 19]	APH [9]	BDD [21] DPLL [7]	I/O-IMC [5]
Pareto fronts	BU [22, 19]	OPEN PROBLEM	$\mathcal{C}$-BU [11]	PTA [8]
Any of the above	**Algo. 1:** $\mathtt{BU}_{\mathtt{SAT}}$	**Algo. 5:** $\mathtt{BU}_{\mathtt{DAT}}$	**Algo. 2:** $\mathtt{BDD}_{\mathtt{DAG}}$	OPEN PROBLEM
***k*-top metrics**	BU-projection [14]	OPEN PROBLEM	**Algo. 3:** `BDD shortest_paths`	OPEN PROBLEM

4 APPROACH AND METHODOLOGY

In order to be able to answer the Research Question(s) introduced above as conclusively as possible, this research will be data-driven and with a correlational design. This paper's focal point is to understand the correlation between different structural properties of the trees and the computation time of those trees' metrics. In order to do that, we must be able to measure and define the parameters of each tree accurately. We must also accurately measure the computation time, which is a trivial issue. The approach to answering the

Research Question can be summarised into four main phases: tree generation, conversion from AT to BDD, calculation of metrics, and finally, the analysis of the data.

The first phase is about generating a diverse corpus of ATs, where the graphical properties of the ATs of said corpus span a large number of values. For example, when looking at the number of nodes, we want to generate a corpus containing different ATs with an increasing number of nodes (e.g., starting 10 nodes up to 1000 nodes). This phase is needed given the sensitive nature of ATs; there is no established benchmark AT due to confidentiality reasons.

The second phase concerns converting each AT into its BDD form. The ATs are generated using the *NetworkX* library, while the BDDs use the *Sylvan* library. In order to convert the NetworkX graph into a BDD, we first extract the Boolean formula that represents the given AT (see Algorithm 2). We then use that Boolean expression as input to *Sylvan*, which automatically generates the corresponding BDD. This BDD is what will be used for the calculation of the security metric. To calculate the security metric, we use the aforementioned BDD as input to the algorithm mentioned in Section 4.4.

4.1 Tools

A library is needed for graph creation and manipulation, which will be most useful when populating the AT corpus on which the graph metrics will be calculated. We have decided to use FOSS (Free Open-Source Software) for this due to its transparent nature, which gives us the possibility of adapting the code to our own needs. In order to find and choose a library, many available libraries are considered, and the best fit is selected. This selection is based on three unordered criteria:

(1) *Intended application.* It should be taken into consideration the intended use of each library. A library developed specifically for Attack Trees would have an advantage over a library developed for tree manipulation.
(2) *Programming language.* The language for which the library has been developed is an important factor to consider. The preferred programming language is Python due to its simple syntax, powerful functionality, and third-party libraries' availability. However, other programming languages could be considered if needed, such as C++, Java, OCaml, Matlab, or even Haskell.
(3) *Latest commit.* Libraries that are no longer being actively supported should be avoided. If a library's latest commit was years ago, it signifies that the developers have stopped developing it, and there will be minimal support.

ADTGenerator. ADTGenerator [7] is a command-line tool that generates Attack-Defence Trees (ADTs). Unfortunately, it falls short on two of the three previously mentioned criteria. The last commit to the repository was in June of 2019, almost three years ago. Furthermore, it generates ADTs, not Attack Trees. Even though ADTs are basically Attack Trees but extended with extra steps that could be taken to stop the attacks, we only need ATs for this paper. Having ADT introduces the extra step of "reducing" them into ATs, which should be avoided if possible.

ADTool. ADTool [19] is a tool that allows users to manipulate and visualise ADTs and perform quantitative analyses on them. Even though ADTool supports Attack Trees, it still has a glaring issue. The last commit to the repository was in 2017, five years ago. Even though we may only need some basic functionality that may already be fully functional, we decided not to use this library in case we ran into any issues, at which point we would be stuck and have limited support.

attackTrees. The goal of attackTrees [9] is to model, analyse, and render attack trees. At first glance, this library meets all three criteria mentioned; it is explicitly developed for attack trees and intended to be used with Python. The latest commit (at the time of writing) was in September 2021, eight months ago. However, there is a clear Work In Progress warning, with a disclaimer from the developer that it is not recommended for anyone to use this. Furthermore, it was brought to light that the Attack Trees the library aims to model are a completely different concept but happen to have the same name.

mindmup-as-attack-trees. This library [3] aims to utilise the mindmup platform and its framework as a medium to manipulate and analyse Attack Trees. The tool uses scripts written in Python to manipulate the given mindmup graphs. This is an advantage since those scripts can be modified if necessary. However, using the mindmup platform introduces more work and possible points of failure since we would have to convert to and from mindmup. The latest commit at the time of writing was in August 2021 (8 months ago).

NetworkX. NetworkX [14] is a Python library that allows for the creation, manipulation, and analysis of complex networks. The most significant advantage of that library is its popularity and, by extension, its support. Its modularity and flexibility are double-edged; it gives the user much control over the graphs, but that also means that there will not be any AT-specific functions, thus introducing an extra layer of work. The latest commit on the repository was 16 hours ago at the time of writing, so it is safe to assume that it is still being actively supported.

Sylvan. Sylvan [20] is a library that is focused on BDDs, written in C, for C/C++ development. It implements parallelised operations on BDDs and supports "any kind of terminal, including standard Booleans, integers, floating points, and any user-defined types" [20]. Sylvan also has bindings for languages other than C/C++, including Python. This can be found in the library called "dd" [10].

4.1.1 *Selection/Analysis.*

After considering each library mentioned above, two libraries have been chosen to be used throughout this project: NetworkX and Sylvan/dd. The reason that NetworkX was chosen is that it meets almost all three of the mentioned criteria. It is made for Python, which is favourable. It arguably has the most robust support of all the other libraries, and even though it is not explicitly developed for ATs, it has strong support for DAGs.

Furthermore, Sylvan/dd was also chosen for its high relevance to our criteria. Its intended application is almost identical to our intended use, and while it may not have been developed for Python, it has bindings for other languages, including Python. Finally, even though it may not have strong support, its primary author Tom van Dijk, who is currently an assistant professor at the Formal Methods and Tools group at the University of Twente. This is the same group where the current work is being developed, thus the choice of Sylvan is expected to greatly facilitate support in case of technical difficulties with the usage of the tool.

4.2 Tree Generation

Since ATs represent sensitive information about a given system, it would be tough to find any real-life examples. As such, we will generate our own pool of random ATs, where each tree should be somewhat similar to a real-life AT. In order to ensure that our generated pool holds some similarity to real-life ATs, we will use "seeds", which are taken from different sources [12] [8] [11] [1] [2]. These seeds would then be used as building blocks, where we would combine them in random ways in order to construct bigger ATs.

The first step in generating our AT pool is first to get our seeds. The structures of the ATs—as taken from their original sources—that are to be used as seeds, were hard-coded as constants into the code of the project, to make its continuous reuse as efficient as possible. The next step is to find ways to combine any two given ATs. To that end, two methods of combination were defined and implemented: adding as a leaf and adding as a sibling.

4.2.1 Adding as a leaf. Given two ATs g_1 and g_2, rooted at r_1 and r_2 respectively, this method aims to append g_2 as a child of some leaf node in g_1. For example, consider Figure 3a, where we have two disjoint graphs g_1 and g_2. A first implementation of adding g_1 as a leaf of g_2 is shown in Figure 3b. In a mathematical/graphical framework, Figure 3b shows a correct implementation. However, in the context of an AT, having a node a with only one child b does not make sense. If it only takes node b to activate node a, then from a logical perspective, the two nodes could be considered to be a single node. Each Intermediate Event node represents a logical function (namely AND and OR), and its children represent the arguments of its function. Since the functions both need at least two arguments, then it follows that each Intermediate Node must have at least two children nodes.

To resolve this issue, a new node with random metrics is added as a sibling to r_2, such that the former leaf node of g_1 has at least two children. This can be seen in Figure 3c.

4.2.2 Adding as a sibling. Given two ATs g_1 and g_2, rooted at r_1 and r_2 respectively, this method aims to join the two graphs such that they both share the same parent r_3. For example, consider the two graphs given in Figure 3a. Joining them as siblings would result in a single graph g_3 rooted at r_3, as seen in Figure 3d.

Given those two methods of combining two given graphs, along with a pool of graphs taken from the literature, we were able to generate graphs by recursively calling the two methods. The generation function has four parameters of interest that should be pointed out. The first two are the two graphs that will be combined. The third parameter, *base_prob*, indicates the probability of making a recursive call. The fourth parameter, *prob_factor*, is the factor by which we multiply *base_prob*, whose product will become the next iteration's *base_prob*. See Algorithm 1 for the pseudo-code of the tree generation algorithm.

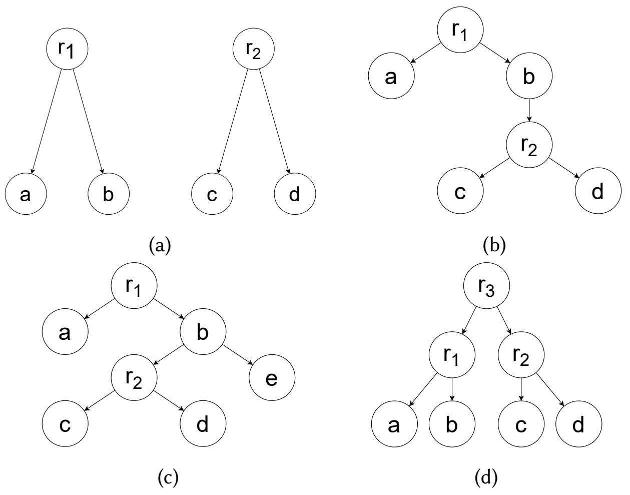


Fig. 3. Adding two subgraphs

Finally, given the final generated graph, the last step is generating and attaching the labels and metrics. An alphabetic label is attached to each leaf node (BAS), as well as a random (integer) metric between 0 and 100.

Algorithm 1 tree_generation

```
Input: $g_1$, $g_2$, base_prob: float, prob_factor: float, min_iters: int
Output: $g_3$: Graph
 1: $p \leftarrow uniform\{0..1\}$          ▷ Get random number between 0 and 1
 2: $h \leftarrow binary\{0, 1\}$                ▷ Randomly pick 0 or 1
 3: if $h = 1$ then
 4:     $g_3 \leftarrow add\_as\_sibling(g_1, g_2)$
 5: else
 6:     $g_3 \leftarrow add\_as\_leaf(g_1, g_2)$
 7: end if
 8: $g_4 \leftarrow pick\_random\_seed()$          ▷ Pick a random seed graph
 9: if $min\_iters > 0$ then
10:     return tree_generation($g_3, g_4, base\_prob, prob\_factor, min\_iters-1$)
11: end if
12: if $p \leq base\_prob$ then
13:     $new\_prob \leftarrow base\_prob \times prob\_factor$
14:     return tree_generation($g_3, g_4, new\_prob, prob\_factor, min\_iters-1$)
15: else
16:     return $g_3$
17: end if
```

4.3 Conversion from DAG to BDD

As shown in [6], any AT could be transformed into a BDD while conserving each BAS's label and metric. Given that BDD, one could perform the analysis on it in a bottom-up fashion and end up with correct results. This conversion could potentially save much time since - despite BDDs being DAG-structured as well - bottom-up algorithms work much better on BDDs [6]. As such, we decided that any ATs that will be analysed will first be converted into BDDs, even if they are tree-structured. The reason for this is twofold; to provide an equal comparison between all ATs and keep the bottom-up algorithm simple.

As discussed in Section 4.1, we use the Sylvan library and its Python bindings to manage BDDs. The dd library, which is the Python binding of Sylvan, needs a Boolean formula as input to generate the corresponding BDD. To that end, given an AT, we needed the Boolean expression that represents that AT, where the BASs are the formula's variables. In order to obtain the formula, a recursive top-down approach was used. See Algorithm 2 for the pseudo-code. Note that in line 7, the *join* function returns a string, which is obtained by combining each element in *children* and inserting the *gate* character between each element.

Algorithm 2 get_boolean_from_AT

```
Input: g: Graph, root: Node
Output: expression: String
 1: if root is not leaf then
 2:     children ← []
 3:     gate ← node.gate
 4:     for node in root.children do
 5:         children.append(get_boolean_from_AT(g, node))
 6:     end for
 7:     return gate.join(children)
 8: else
 9:     return node.label
10: end if
```

4.4 Calculating metrics

Considering the scope of this paper, it was decided only to consider the calculation of one metric, namely the minimum cost of an attack. We are not interested in which metric is being calculated but rather how long that calculation takes, so the minimum cost metric was arbitrarily chosen. As such, the implementation of that bottom-up (BU) algorithm is based on Algorithm 2 from [6]. Note that this algorithm was developed for general metrics; thus, the calculation time should not be affected by any specific metric.

4.5 Data collection and visualisation

In order to reliably answer each Research Question and to remove as much uncertainty from the results as possible, we have employed some redundancy in the generation of the trees, as well as in the collection of runtime data. For each sub-question (with some exceptions), the first step is to define at least ten different data points that we will want to look at. For example, when considering an AT's size (number of nodes), we will want to look at ATs with (10, 20, 30, 50, 100,...) nodes. Those will also be the ticks on the x-axis.

Then, for each data point, we will generate three different ATs that meet that data point's criterion. The reason we will use three different ATs is to try to minimise any ambiguity concerning the causality of the results. In other words, we want to ensure that if there were to be a correlation between our data points and the runtimes, then that correlation is because of the property we are observing, not some other unobserved property.

Finally, for each of the generated ATs, we will run the BU algorithm ten times and record the runtime of each iteration. Note that we only record the runtime of the BU algorithm on the BDD representation of an AT, and we *do not* include the time taken to generate that AT or to convert it into a BDD.

We will use vertical box plots to represent each AT's runtimes to plot the data. Since we have generated three different ATs per x-tick, there will be three box plots per column on the graph. To avoid confusion, each of the three plots will be coloured differently. Furthermore, we will omit the outlier circles (or 'fliers') as they clutter the graphs and do not convey relevant information.

4.5.1 *Some exceptions.*

In some cases, it is hard to generate an AT that meets a specific criterion, as it would take too long to generate one, given the random nature of the generation algorithm. Likewise, it would take too much time to alter the parameters of said algorithm. As such, a different approach was employed. Instead of first defining at least ten data points to observe, we will randomly generate many ATs. For each of those ATs, we will calculate the property being observed (i.e., size, number of foster nodes). The current AT will be discarded if an AT has previously been generated with the same calculated property. This is to avoid having duplicate x-ticks. From there, the BU algorithm is run ten times on each of those ATs, and their box plots will be graphed. This approach was used when answering questions 3, 5, 6, and 7.

5 RESULTS

In this section, we will attempt to answer each sub-question defined in Section 2.1. For Sections 5.1, 5.2, and 5.4, note that on each column, there are three separate box plots, each with a different colour, representing the ten run-times on that AT. They are colour-coded such that the box with the largest median is blue, the box with the lowest median is red, and the box with a median in between the other two is green. Some of them may appear as horizontal lines, but that is because all ten runtimes are quite close to each other, especially compared to the scale of the y-axis. For Sections 5.3, 5.5, 5.6, and 5.7, each column only has one box plot. This is because the alternative approach (discussed in Section 4.5.1) was employed.

In some cases, many of the values are very close to 0, while a few are much higher (see Figures 5a and 13a). Due to the fact that the y-axis is on a linear scale, it becomes challenging to read the graphs. To that end, we used a logarithmic scale on the y-axis when necessary (see Figures 5b and 13b).

For each subsection, in order to maintain objectivity, we calculate the correlation coefficient (Pearson's r-value for measuring linear correlations) between the x- and y-values. For any graphs with three points per x-value (the medians of each box plot), we use the mean of those three points as our x-values when calculating the "Average r-value." We also calculate the r-values of each colour for completeness. For the sections that employ the approach discussed in Section 4.5.1, we simply use the medians of each box plot as our x-values when calculating the r-value.

According to [16], the calculated r-values should be interpreted as such:

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

Note that these ranges are the absolute values of the r-values.

5.1 Size

In this section, we will attempt to answer sub-question 1; "How does the size (number of nodes) of the AT affect the computation time of its metrics?". As can be observed in Figure 5a, there appears to be no correlation between the size of an AT and the runtime of the analysis of that AT. Looking at Figure 5b, even with a logarithmic scale, there is no apparent correlation between the size and runtime.

This lack of correlation is supported by the following r-values:

- Red: r = -0.1975815932189488
- Green: r = -0.20938905844438085
- Blue: r = -0.06487039829335528
- Average r-value: r = -0.0684892858684561

Given these r-values, we can claim that, given the data collected, there is no significant linear correlation between the size of an AT and the analysis runtime of that AT.

5.2 Out-degree

Considering the second sub-question, we have generated different ATs whose nodes have specific out-degrees. The criterion that each AT has to meet is that each non-leaf node in said AT should have an out-degree of at least **x**. If needed, we add leaves to said node until it meets the required criterion.

As shown in Figure 6, there appears to be a slight negative correlation between the out-degree of the nodes and the run-times. Note that the y axis is on a logarithmic scale. This is also supported by the r-values of each color:

- Red: r = -0.33278300073698297
- Green: r = -0.33994692873285637
- Blue: r = -0.3399259357792736
- Average r-value: r = -0.3395127136620615

These r-values suggest a weak negative correlation between the out-degree of an AT's nodes and the runtime of the analysis algorithm. While this is an unexpected result, one could speculate that this is due to the generation process for this subsection. In order to generate ATs whose nodes have a specific number of children, we manipulate each generated AT to meet the specific criterion.

For example, if we want to generate an AT whose nodes have an out-degree of 12, then we generate a random AT and go through each of that AT's nodes. For each node, as long as it has an out-degree less than 12, we add a leaf to that node. While this approach is simple, it may have affected the above results. We speculate that this excess addition of BAS nodes to the AT results in a large graph with many leaves.

This large number of leaves would translate to a large number of variables when we extract the Boolean formula of that AT. Consequently, because many of those variables were simply added as leaves, they probably have not affected the underlying logical expression of the AT. As a result, despite having many variables in the Boolean formula, the generated BDD is quite small. The reason for that is because the `dd` library outputs *reduced ordered BDDs* [10]. Having a reduced BDD means that "given variable order, equivalent propositional formulas are represented by a unique diagram" [10]. In other words, all the extra BAS nodes become increasingly redundant, which results in a smaller BDD. This is under the assumption that the variable ordering of the BDD is not in the worst case.

5.3 In-degree

To answer the third sub-question, we had to look at the number of parents of the nodes in an AT. Since ATs are DAG-structured, nodes could have multiple parents. As such, we specifically look at the mean in-degree of the nodes of each AT. Since it would be too difficult and time-consuming to generate ATs with a specific mean in-degree, we have taken the alternative approach discussed in Section 4.5.1.

As shown in Figure 7, there seems to be no correlation between the mean in-degree of the nodes and the runtime of the analysis. Note that two values were excluded from the graph, namely the values at *x=1.039* and *x=1.042*. Those were omitted as they are considered outliers and, given the linear scale of the y-axis, rendered the graph much harder to read. It should be noted that the logarithmic scaling of the y-axis did not demonstrate any suggestions of a correlation either.

Again, this is supported by the r-value of the graph. When calculated, it gave a value of *r=-0.22756249154669397*. This indicates a weak negative correlation between the average in-degree of an AT's nodes and the runtime of its analysis. Note that the outliers were included in the calculation of the r-value.

5.4 Depth

As depicted in Figure 8, there are no apparent correlations demonstrated. This is demonstrated when looking at the calculated r-values:

- Red: r = -0.3112599051785191
- Green: r = -0.22321313273558854
- Blue: r = -0.09395435570178053
- Average: r = -0.16721945634502614

Note the Average r-value indicating a weak negative correlation that is almost negligible [16].

5.5 Root to foster nodes

This section considers the median of the depths of all foster nodes of a given AT. The depth of a node is defined as the maximal shortest distance between the root of the AT and the node. As demonstrated in Figure 9, there seems to be no clear correlation between the property and the algorithm's runtime. Note that, due to the randomness of the generation algorithm, we have employed once again the alternative approach mentioned in Section 4.5.1. Furthermore, we've excluded the points at *x=3.0* and *x=5.5*, for the same reasons mentioned in Section 5.3. This lack of correlation is also supported by the r-value of *r = -0.2964100141122183*, which indicates a weak negative correlation.

5.6 Foster nodes to leaves

In this section, we consider what could be called the "opposite" of the previous property. That is, we consider the median of distances between each foster node in an AT and the leaves of that AT. The distance between each foster node and the AT's leaves is calculated by finding the maximal shortest distance between said node and all the children that are reachable from that node.

According to Figure 11a, there seems to be no clear correlation between the graph property in question and the runtime of the analysis algorithm. Not that the value for *x=6* and *x=8.5* have been excluded due to their being outliers. Given the general low values, Figure 11b shows the same data on a logarithmic scale for the y-axis. However, despite the wider vertical distribution, there is seemingly no correlation between the graph property in question and the algorithm's runtime. This is supported by the calculated r-value of *r = 0.11583709596444068*.

5.7 Number of foster nodes

As shown in Figure 13a, there is a strong increase in the run-times starting at *x=71*. However, all the box plots before that are close to 0, with some slight variance. In order to be able to see those subtle differences clearly, the same graph has been plotted with a logarithmic scale on the y-axis in Figure **??**. Observing Figure 13b, there is a very clear trend upwards, which is supported by the highest r-value so far of *r = 0.5106236999548139*. Indicating a Moderate Correlation, this strongly suggests a direct positive correlation between the number of foster nodes in an AT and the runtimes of the analysis algorithm.

It is important to remember that this apparent correlation is visible when the y-axis is on a logarithmic scale. This means that the correlation between the number of foster nodes and the runtimes is not linear, but logarithmic.

6 CONCLUSION

Graphical Property	Pearson's r-value
Number of foster nodes	0.5106236999548139
Out-degree	-0.3395127136620615
Root to foster nodes (median)	-0.2964100141122183
In-degree	-0.22756249154669397
Depth	-0.16721945634502614
Foster nodes to leaves (median)	0.11583709596444068
Size	-0.0684892858684561

Table 2. An overview of the results

Table 2 shows the calculated r-value of each graphical property we have considered throughout this paper. They are sorted by descending (absolute) r-value. The graphical property affecting AT analysis's time complexity most is the number of foster nodes present in said AT. This is supported by the r-value shown in the table, which is significantly higher than all other r-values.

Furthermore, there seem to be many weak **negative** correlations between the different graphical properties and the runtime of the analysis. The author could not find a clear and definitive explanation for these results. While some possible explanations have been discussed in the relevant sections, those are nevertheless speculations.

Notwithstanding, it is vital to bring to the reader's attention that much of the author's implementations of the discussed algorithms have not been peer-reviewed. To wit, one must not rule out the possibility that there may be logical bugs and discrepancies in the code used throughout this project. The extent of the effect of such potential bugs is hard to gauge, as it could be a simple, negligible issue or a fundamentally flawed logical error. However, one could argue that given the presented results, it would be difficult to argue that the author's implementations are fundamentally wrong, or else one would not end up with such precise and consistent results.

6.1 Future Work

In hindsight, there is a possibly better approach to the Tree Generation phase of this project. Instead of generating a different corpus of random ATs for each graph property that was explored in this paper, a better approach would have been to generate one large corpus of random ATs. We would then use this large pool for each graph property and keep track of which graphs were used and the results of each graph.

Another significant advantage of this approach is that we will be able to cross-reference each AT between the different observed properties. In other words, we would be able to cross-check conjectures. For example, if the number of foster nodes matters, but not the total number of nodes, then we would be able to see that those ATs with a high number of foster nodes would have the largest runtimes across all graphs.

This approach implies that we must be able to uniquely identify each AT in the pool, which is possible since the *NetworkX.DiGraph* object is hashable. This means we could easily store each AT in a dictionary or map, along with each AT's important properties. Such a comprehensive and organised data structure could also open doors to different, possibly better approaches.

For example, given a structure where each AT is stored along with its different values for each of the seven graph properties explored in this paper, one could employ Machine Learning. More specifically, by using Feature Importance techniques, we may gain deeper insights, as well as much more accurate results [21].

Finally, one last addition to this paper that we, unfortunately, could not include due to time constraints is also including the time taken by the conversion phase. Throughout this paper, we have not considered the time it takes to convert an AT to a BDD, as we wanted to focus on the runtime of the metric calculation algorithm. However, in real life, in order to obtain a metric given an AT (using this paper's approach), then the conversion time would indeed be a part of the process and a significant one. To that end, we believe that exciting insights can be gained on the conversion times and what graph properties affect that.

REFERENCES

[1] Florian Arnold, Dennis Guck, Rajesh Kumar, and Mariëlle Stoelinga. 2015. Sequential and Parallel Attack Tree Modelling. In *Computer Safety, Reliability, and Security (Lecture Notes in Computer Science)*, Floor Koornneef and Coen van Gulijk (Eds.). Springer, Netherlands, 291–299. https://doi.org/10.1007/978-3-319-24249-1_25 Foreground = 80Type of audience = scientific community; Size of audience = 30; Countries addressed = international;; 34th International Conference on Computer

Safety, Reliability, and Security, SAFECOMP 2015, SAFECOMP ; Conference date: 22-09-2015 Through 22-09-2015.

[2] Florian Arnold, Holger Hermanns, Reza Pulungan, and Mariëlle Stoelinga. 2014. Time-dependent analysis of attacks. In *Proceedings of the Third International Conference on Principles and Security of Trust, POST 2014 (Lecture Notes in Computer Science)*. Springer, Netherlands, 285–305. https://doi.org/10.1007/978-3-642-54792-8_16 Foreground = 50% ; Type of activity = Conference ; Main leader = UT ; Type of audience = scientific community ; Size of audience = 100 ; Countries addressed = international ;; 3rd International Conference on Principles and Security of Trust, POST 2014, POST ; Conference date: 05-04-2014 Through 13-04-2014.

[3] BenGardiner. 2021. mindmup-as-attack-trees. https://github.com/BenGardiner/mindmup-as-attack-trees

[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] Carlos E. Budde, Christina Kolb, and Mariëlle Stoelinga. 2021. Attack Trees vs. Fault Trees: Two Sides of the Same Coin from Different Currencies. In *Quantitative Evaluation of Systems*, Alessandro Abate and Andrea Marin (Eds.). Springer International Publishing, Cham, 457–467.

[6] Carlos E. Budde and Mariëlle Stoelinga. 2021. Efficient Algorithms for Quantitative Attack Tree Analysis. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–15. https://doi.org/10.1109/CSF51468.2021.00041

[7] Matthias Eckhart, Kristof Meixner, Dietmar Winkler, and Andreas Ekelhart. 2019. Securing the testing process for industrial automation software. *Computers & Security* 85 (2019), 156 – 180. https://doi.org/10.1016/j.cose.2019.04.016

[8] Marlon Fraile, Margaret Ford, Olga Gadyatskaya, Rajesh Kumar, Mariëlle Ida Antoinette Stoelinga, and Rolando Trujillo-Rasua. 2016. Using attack-defense trees to analyze threats and countermeasures in an ATM: A case study. In *9th IFIP WG 8.1 Working Conference on The Practice of Enterprise Modeling (PoEM) (Lecture Notes in Business Information Processing)*. Springer, Netherlands, 326–334. https://doi.org/10.1007/978-3-319-48393-1_24 Foreground = 100Type of audience = scientific community; Size of audience = 25; Countries addressed = international;; null ; Conference date: 08-11-2016 Through 10-11-2016.

[9] hyakuhei. 2021. attackTrees. https://github.com/hyakuhei/attackTrees

[10] johnyf. 2022. dd. https://github.com/tulip-control/dd

[11] Barbara Kordy and Wojciech Wideł. 2018. *On Quantitative Analysis of Attack–Defense Trees with Repeated Labels.* 325–346. https://doi.org/10.1007/978-3-319-89722-6_14

[12] Rajesh Kumar, Enno Ruijters, and Mariëlle Stoelinga. 2015. Quantitative Attack Tree Analysis via Priced Timed Automata, Vol. 9268. 156–171. https://doi.org/10.1007/978-3-319-22975-1_11

[13] Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. 2020. A review of attack graph and attack tree visual syntax in cyber security. *Computer Science Review* 35 (2020), 100219. https://doi.org/10.1016/j.cosrev.2019.100219

[14] NetworkX. 2022. NetworkX. https://github.com/networkx/networkx

[15] B. Schneier. 1999. Attack Trees. *Dr. Dobb's Journal* (1999).

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

[17] Chris Slater, O. Saydjari, Bruce Schneier, and Jim Wallner. 1998. Toward a Secure System Engineering Methodolgy. 2–10. https://doi.org/10.1145/310889.310900

[18] T. Sonderen. 2019. A Manual for Attack Trees. http://essay.utwente.nl/79133/

[19] tahti. 2017. ADTool2. https://github.com/tahti/ADTool2

[20] Tom van Dijk. 2022. Sylvan. https://github.com/trolando/sylvan

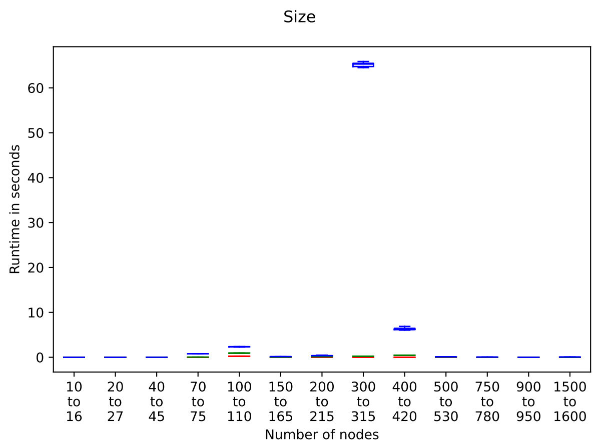
[21] Alexander Zien, Nicole Krämer, Sören Sonnenburg, and Gunnar Rätsch. 2009. The Feature Importance Ranking Measure. In *Machine Learning and Knowledge Discovery in Databases*, Wray Buntine, Marko Grobelnik, Dunja Mladenić, and John Shawe-Taylor (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 694–709.

A PLOTS

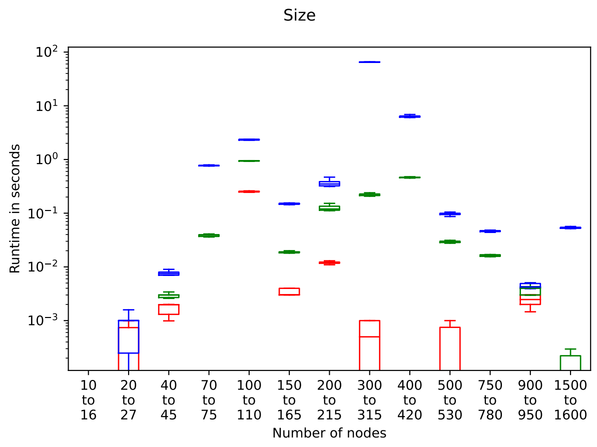
A.1 Size

Fig. 4. Size

(a) Size on linear y-scale

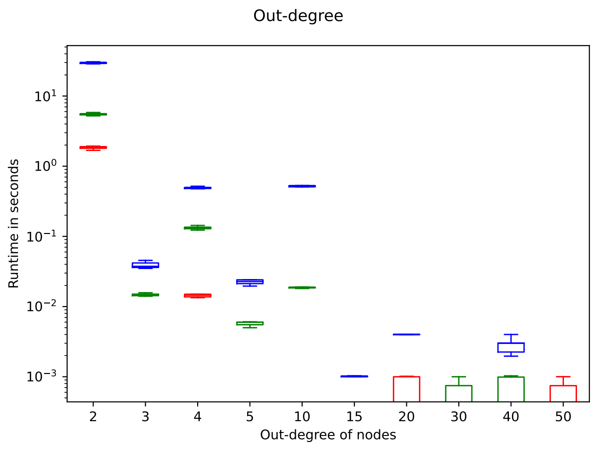


(b) Size on logarithmic y-scale



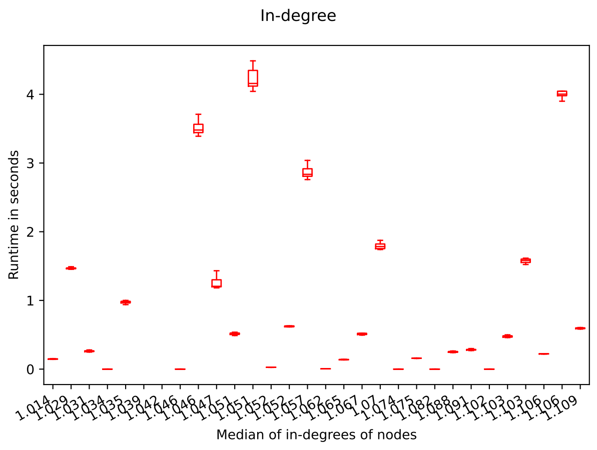
A.2 Out-degree

Fig. 6. Out-degree on logarithmic y-scale



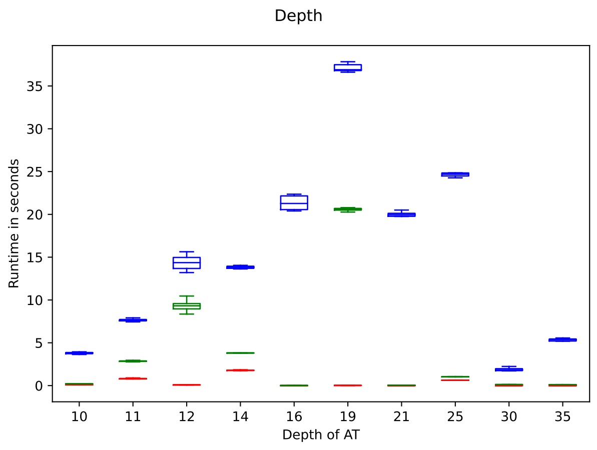
A.3 In-degree

Fig. 7. In-degree on linear y-scale



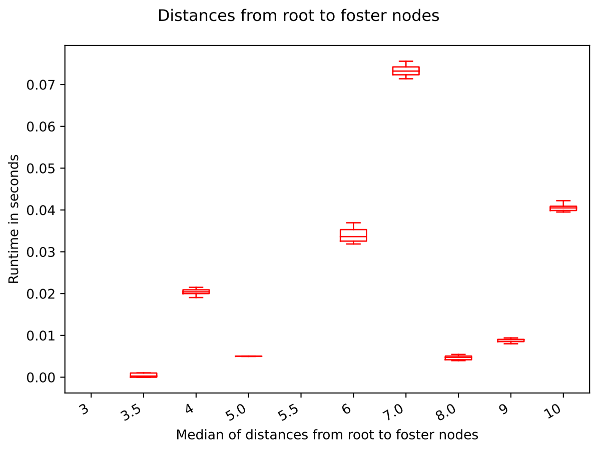
A.4 Depth

Fig. 8. Depth on linear y-scale



A.5 Root to foster nodes

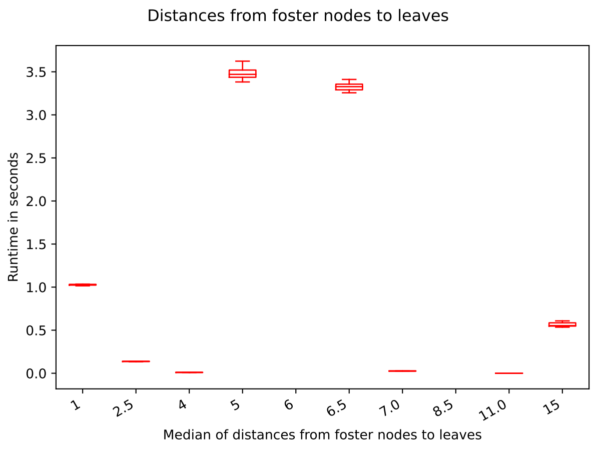
Fig. 9. Root to foster nodes on linear y-scale



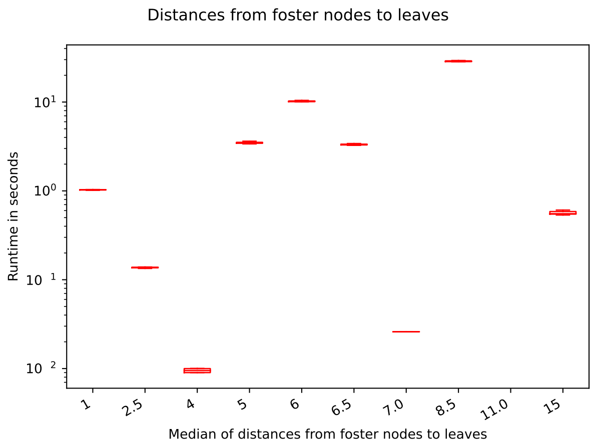
A.6 Foster nodes to leaves

Fig. 10. Foster nodes to leaves

(a) Foster nodes to leaves on linear y-scale



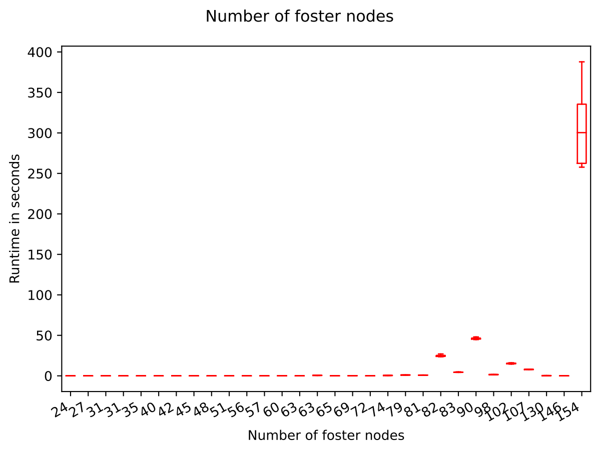
(b) Foster nodes to leaves on logarithmic y-scale



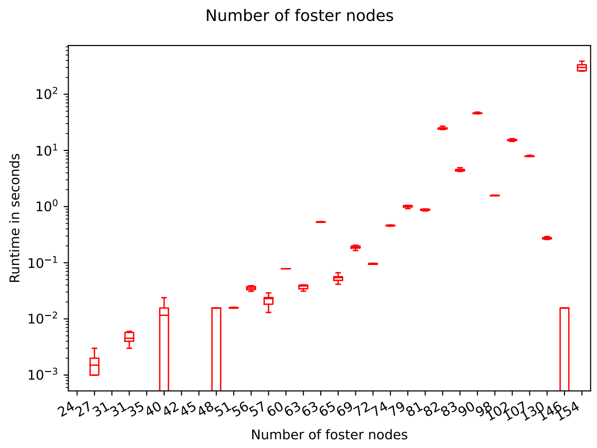
A.7 Number of foster nodes

Fig. 12. Number of nodes

(a) Number of foster nodes on linear y-scale



(b) Number of foster nodes on logarithmic y-scale


Number of foster nodes
Runtime in seconds
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
Number of foster nodes