Crunching Attack Trees: Efficient Algorithms for Computation of Metrics

Jarik Karsten
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.g.karsten@student.utwente.nl

ABSTRACT

Attack Trees are used to analyze system attacks. Their analysis provides security metrics that can be used as a resource to protect systems and increase their security levels. As the complexity of attacks and systems increases, Attack Trees become complex by incorporating DAG structures and attack orderings. This paper develops and implements algorithms to compute important security metrics such as Attack Values, Attack Paths, and Pareto Curves compared to existing model checking solutions, with EXP-complexity as opposed to the PSPACE-complexity of the current state-of-the-art. We achieve these complexity bounds by using efficient BDD data structures and abstractions regarding the total orderings of basic attack steps.

Keywords

Attack Trees, Attack Values, Attack Paths, Pareto Curves

1. INTRODUCTION

As the security of computer systems become increasingly complex, frameworks and standards such as Attack Graphs [22] were introduced to model and analyze these systems. Attack trees, first introduced by Weiss [23], and popularized by Schneier [21], provide analysis by computing potential attacks and their respective values such as cost and time.

1.1 Attack Trees (ATs)

ATs capture all possible attacks in a tree structure, which features three types of nodes.

1. *Basic Attack Step (BAS)*: The leaf nodes represent actions that are not split up further with certain attributes such as cost and time.
2. *Gates / Connectors*: AND/OR-gates show how attacks propagate and indicate whether, respectively, all or at least one child should succeed.
3. *Root*: The top node is the root of the AT and the goal of the attack.

An example of such an AT is provided in figure 1 from [13]. To *Obtain Administrator Privileges* an attack has to either complete *Access System Console* or *Obtain Administrator Password* as indicated by the OR-gate. These gates are further refined until we reach the leaf nodes such as *Break Into Computing Center* on the bottom left, which alone would compromise the root node as its parents are all OR-gates.

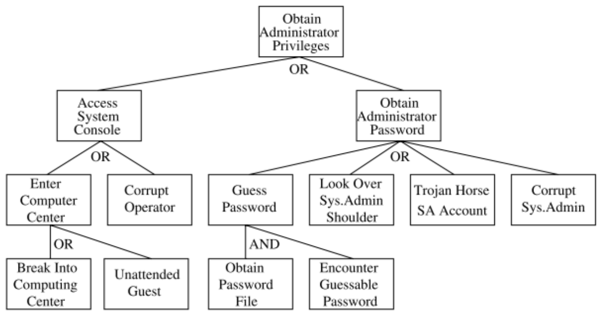


Figure 1: AT for obtaining Administrator Privileges [13]

1.2 Definitions

For ATs, an attack consists of a set of BASs. We introduce three terms that are relevant in the analysis of these attacks.

1. *Attack Values* refer to the values of the cost and time of an attack.
2. *Attack Paths* refer to the BASs involved in the attack, and the order of these *BASs* if this is relevant.
3. *Pareto Curves* analyze trade-offs between Attack Values. In this study, we focus on the trade-offs between cost and time. These trade-offs can provide insights into how attack costs decline with additional time.

We remark that there is a distinction between sequential and parallel attacks. Parallel attacks allow multiple BASs to execute in parallel, while sequential attacks execute them sequentially. This paper will discuss parallel attacks, but sequential attacks are analyzed similarly with asymptotically equal or lower complexity. The appendix contains supplementary information to achieve this by discussing how to adjust the algorithm to use sequential attacks.

1.3 Extensions

As attacks became increasingly complex, DAGs and attack orderings were introduced for ATs. This domain currently does not have efficient analysis metrics for attacks, as introduced in section 1.2, which this paper aims to improve. These new concepts are defined as follows:

- *Directed Acyclic Graphs (DAGs)*: By allowing nodes to share subtrees, ATs become DAGs instead of trees.

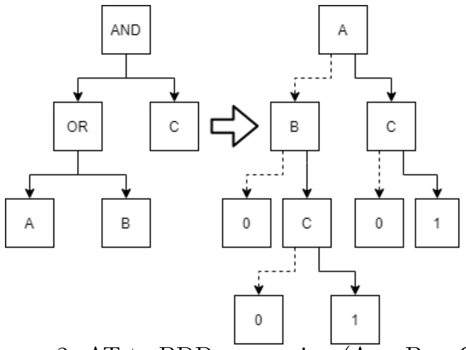


Figure 2: AT to BDD conversion (A < B < C)

- *Order of attacks*: New gates were proposed [5] to indicate a particular order that the children are triggered in. They specify that certain BASs should be completed before other BASs can start execution. Commonly these gates are referred to as sequential-AND (SAND) or ordered-AND (OAND) gates.

In the field with these additional extensions, it is necessary to calculate minimum value attacks. This paper aims to compute these minimum time/ cost attacks using the following definition of an AT.

Definition 1. Attack Tree (AT): (N, Child, Values, Root).
- N: The finite set of nodes of the AT.
- $Child$: $V \to V^*$ maps each node to it's child nodes creating a DAG.
- $Values$: $V \to (cost \in \mathbb{R}_+, time \in \mathbb{R}_+)$ maps BASs to their cost and time.
- $Root \in N$: Root of the AT and goal of the attacker.

1.4 Related Work

Several tools have been created, such as AttackTree+ [11] and SecurITree [4], which can be efficient, but they frequently cannot handle the ordering of events, shared subtrees, and constraints. Algorithms to solve Attack Values, Attack Paths, and Pareto Curves were discussed in title [16]. The paper demonstrates how ATs can be translated into *Priced Timed Automata* [8], an extension of *Timed Automata* [3]. Previous work has shown that model-checking programs such as UPPAAL can find these Attack Values, Attack Paths, and Pareto Curves in various domains [16]. However, these algorithms in model-checking have a PSPACE-complexity [2], which requires a traversal of the entire state space.

2. OPEN RESEARCH DOMAINS

Table 1 provides an overview of the methods for ATs. Bottom-up approaches are usually employed in the case of trees [9], and they can provide linear-time solutions. As an example for time in a parallel environment, MIN(1) for OR-gates and MAX(2) for AND-gates are used in a bottom-up fashion where X/Y denote nodes, and children/time provide the children and time parameters for nodes respectively.

$$\min_{X \in children(Y)} [time(X)] \tag{1}$$

$$\max_{X \in children(Y)} [time(X)] \tag{2}$$

Definition 2. Binary Decision Diagram (BDD) [1]: BDDs are decision trees that encode Boolean functions, where left transitions (branches) negate a variable, and right transitions take the variable. Leaf nodes of the BDD represent Boolean values corresponding to whether the set of transitions satisfy the formula. Figure 2 provides an example of a BDD conversion.

Table 1: Analysis Methods for ATs

AT	Time	Cost	Pareto Curve
Tree	Bottom-up	Bottom-up	[6, 10, 16]
DAG	Bottom-up	BDD	[16]
SAND Tree	[16]	Bottom-up	[16]
SAND DAG	Model checking	Model checking	Model checking

For DAGs, computation of specific attributes is still efficient bottom-up, while others require conversion to BDDs. For instance, efficient computation of cost is no longer viable as BASs can be present in both subtrees. In such cases, costs could be counted twice. With SAND-gates and DAGs, certain areas currently only have PSPACE-complexity algorithms, as introduced in [16].

This research aims to advance the work of [16] by attempting to develop asymptotically lower complexity algorithms, concerning the number of nodes, that improve the analysis of ATs in several research areas. In the analysis, we examine all possible attacks with their corresponding Attack Values and Attack Paths. This leads to the following research questions.

2.1 Research Questions

How can the analysis of sequential Attack Trees with shared subtrees be optimized concerning a wide range of metrics?

1. How can **Attack Values** be efficiently computed for Attack Trees?
2. How can **Attack Paths** be efficiently reconstructed for Attack Trees?
3. How can **Pareto Curves** be efficiently created for Attack Trees?

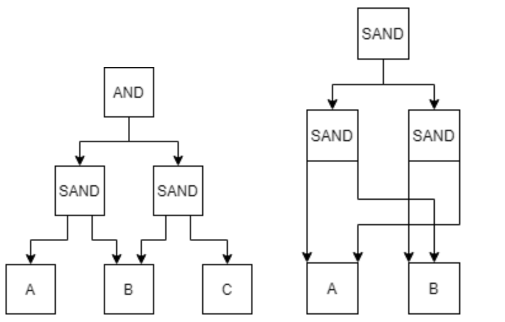
3. DATA STRUCTURES

3.1 BDD

Commonly, BDDs are used in the analysis of ATs. In ATs, it can occur that we have shared subtrees, in which case the AT is a DAG. In such a case determining the cost by combining the left and right subtree is no longer done by simply adding the cost of both as certain BASs might be present in both subtrees. BDDs can convert these ATs into tree structures again, which can simplify the calculation of cost.

However, currently, BDDs do not store temporal dependencies, which are introduced by SAND-gates. To account for this, we should preprocess data or extend the BDD data structure. First, we commence with a concrete definition of SAND-gates, as there have been different interpretations in literature [5, 16].

Definition 3. SAND-gate: Let us define L and R as the sets of BASs present in the left and right subtree of a SAND-gate. The SAND-gate G specifies that each BAS in the left subtree should be completed before BASs in the right subtree can start. However, BASs that are present in both subtrees are an exception. Thus, if we consider (L_G, R_G) for each SAND-gate it should hold that $((l \in L_G) \wedge (r \in R_G \setminus L_G)) \to l < r$, where $l < r$ indicates that l should complete before r starts.



(a) Valid Attack Tree (b) Invalid Attack Tree
Figure 3: Attack Tree Examples

By this definition, figure 3a contains a valid AT and 3b contains an invalid AT.

3.2 Variable Ordering

The SAND-gates impose certain constraints on the partial ordering of BASs. Using topological sort, which takes an Ordering Graph, we construct a total ordering of BASs.

Definition 4. Ordering graph: An Ordering Graph contains a node for each BAS in the AT. Each SAND-gate G has sets of BASs in the left and right tree. Let us denote these by L_G and R_G. The graph will contain a directed edge $(l \to r) \iff (l \in L_G) \wedge (r \in R_G \setminus L_G)$. Here $l \to r$ indicates that l should complete before r starts. If the ordering is not a DAG, we have a cyclic dependency, which makes the AT invalid.

This definition for the Ordering Graph resembles the definition for the SAND-gates. Thus, we can select various SAND-gate definitions as long as the Ordering Graph remains a DAG. If we were to conceive the Ordering Graph for figure 3a, we would have edges $a \to b$ and $b \to c$, which leads to (a, b, c) as a potential total ordering. However, for figure 3b, we have edges $a \to b$ and $b \to a$. In this instance, the graph is no longer a DAG, in which case it is infeasible to obtain a total ordering. In such a case, the AT is invalid.

3.2.1 Topological Sort

To obtain orderings of BASs from the Ordering Graph, we utilize topological sort [18]. Topological sort is used in graph theory to find total orderings of nodes such that if and edge $x \to y$ exists, x occurs before y in the topological order . However, for ATs, additional orderings of BASs might be relevant. For example, (a, b, c) might be a valid ordering, but (b, a, c) could also be a valid and beneficial ordering for the analysis of ATs. We should remark that these additional orderings have the same Attack Values, and only differ in the Attack Path. Thus, they also have no impact on the Pareto Curve.

Tracking these additional orderings during the BDD traversal is not plausible as a solution of n BASs can contain $(n! - 1)$ additional solutions in the worst-case when there are no temporal dependencies. To account for this, we slightly extend the general implementation of the topological sort to reconstruct these solutions after traversal.

For the topological sort, pseudo-code is provided in Algorithm 1. The fundamental concept is that nodes store their incoming edges. In iterations of the nodes, we can eliminate all nodes with no incoming edges, as they have no ordering dependencies. After the removal of these nodes, incoming edges are updated, and the next iteration can commence. This results in sets $A_1, A_2, ...A_n$ where each BAS $x \in A_1$ should occur before each BAS $y \in A_2$ etc. In figure 3a, this would result in the sets $\{a\}$, $\{b\}$ and $\{c\}$.

A pivotal insight to discern here is that we can permute the variables of A_x in any way possible while maintaining the SAND-gate constraints. These permutations allow us to find additional ordering solutions for the AT.

Algorithm 1: Variable Ordering

```
Input: Ordering Graph (DAG)
Output: Sets of BASs used to compute a total
        ordering of BASs
begin
    /* Initialize indegrees of the DAG */
    $n \longleftarrow TotalNodes$ ;
    $indegree \longleftarrow [0] * n$ ;
    foreach $edge\ (x, y) \in DAG$ do
        $inDegree[y] += 1$ ;
    /* Take and remove zero indegree nodes
       until the graph is empty */
    $result \longleftarrow \emptyset$ ;
    $visited \longleftarrow \emptyset$ ;
    $total \longleftarrow 0$ ;
    while $total < n$ do
        $set \longleftarrow nodesIndegreeZero() \setminus visited$ ;
        $total \longleftarrow total + size(set)$ ;
        foreach $edges\ (x, y) \in set$ do
            $indegree[y] \longleftarrow indegree[y] - 1$ ;
        $visited \longleftarrow visited \cup set$ ;
        $result \longleftarrow result \cup \{set\}$ ;
    return $result$
```

3.3 BDD Construction

The BDD requires a total ordering of BASs for construction. The topological sort obtains such a total ordering using the Ordering Graph so that each SAND-gate is satisfied. If the topological sort is unable to obtain such a total ordering, the Ordering Graph contains a cycle which, by definition, makes the AT invalid.

To create the BDD, we utilize ITE structures during construction [12]. These ITE structures define part of the BDD structure, where x is a variable with G and H as child BDD subtrees.

Definition 5. ITE(x, G, H) [12]: If-then-else structure with $(x \to G) \wedge (\neg x \to H)$. x specifies a BAS while G and H are ITE structures or Boolean leaf nodes of the BDD. Satisfying assignments are then given by $(x \cup g)$ and h for satisfying assignments $g \in G$ and $h \in H$.

To create the ITE structure that allows us to translate an AT into a BDD, we have to specify how to create and combine them. The leaf nodes can simply translate into $ITE(x, 1, 0)$, where x is the leaf node's BAS variable.

If we consider an AND-gate in the AT, we construct the ITEs corresponding to the left and right child. Which results in $ITE(x, G_1, G_2)$ and $ITE(y, H_1, H_2)$. Thus, we specify the formulae used in the construction of ITE structures [12], which distinguish between AND/OR-gates and $X = Y(XX)$ and $X \neq Y(XY)$. These formulae depend on non-sequential properties in the AT. As we have defined a total ordering that satisfies the SAND-gates, we can abstract them by replacing them with AND-gates.

$$AND - XX = ITE(x, (G_1H_1 + G_1H_2 + G_2H_1), G_2H_2)$$
$$AND - XY(X < Y) = G_2 * ITE(y, H_1, H_2)$$
$$OR - XX = ITE(x, (G_1 + H_1), (G_2 + H_2))$$
$$OR - XY(X < Y) = ITE(x, G_1, (G_2 + ITE(y, H_1, H_2)))$$

The creation of these ITE-structures can be simplified using Boolean logic to reduce the size of the BDD.

- 0 && ITE = 0
- 1 && ITE = ITE
- 0 && 0 = 0
- 0 && 1 = 0
- 1 && 1 = 1

- 0 || ITE = ITE
- 1 || ITE = 1
- 0 || 0 = 0
- 0 || 1 = 1
- 1 || 1 = 1

4. ALGORITHM

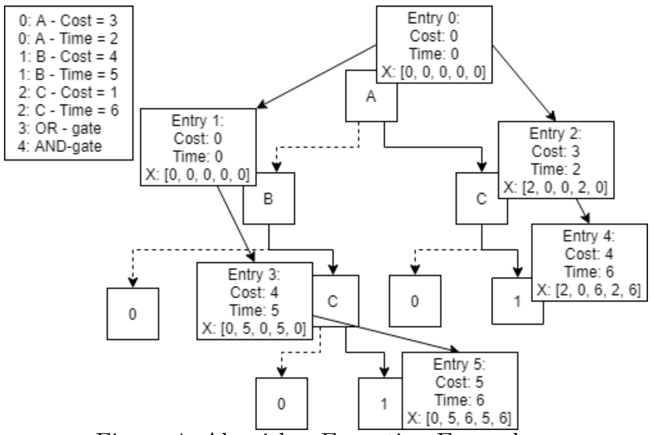
After we determine a total ordering of BASs to construct the BDD, we can traverse the BDD to find Attack Values, Attack Paths, and Pareto Curves. The pseudo-code for the traversal is provided in Algorithm 2, and a sample run is illustrated in figure 4.

During traversal, we create Entries which store possible solutions. An Entry satisfies the AT if its BASs form a successful attack.

Definition 6. Entry (O, C, T, L, X)
- N: The set of nodes of the AT.
- $O \in \{Time, Cost\}$: Ordering used to compare Entries.
- $C \in \mathbb{R}_+$: Current cost of the Entry.
- $T \in \mathbb{R}_+$: Current time of the Entry.
- $L \in \mathbb{N}$: Current location in the BDD.
- $X \in \mathbb{R}_{\mathbb{N}}$: Vector where $X[i]$ denotes required time for completion of node i.

The algorithm uses a central priority queue in which we store Entries sorted on either cost or time, to obtain top-K cost/time attacks. It starts with a single Entry containing the root of the BDD as location. Cost and time are set to zero, and an ordering for the Entries is determined.

In figure 4, we demonstrate an example execution of the algorithm, which takes the BDD of the AT from figure 2. At the start, we have a single empty Entry, Entry 0, in our Priority Queue, and let us assume that we are calculating the minimum cost attacks. We pop Entry 0, and since it is not a leaf, we create two additional Entries corresponding to the left transition with $\neg A$, and the right transition with A. The right Entry 2 gains the cost of A, as it performs



0: A - Cost = 3
0: A - Time = 2
1: B - Cost = 4
1: B - Time = 5
2: C - Cost = 1
2: C - Time = 6
3: OR - gate
4: AND-gate
Entry 0: Cost: 0 Time: 0 X: [0, 0, 0, 0, 0]
A
Entry 1: Cost: 0 Time: 0 X: [0, 0, 0, 0, 0]
Entry 2: Cost: 3 Time: 2 X: [2, 0, 0, 2, 0]
B
C
Entry 3: Cost: 4 Time: 5 X: [0, 5, 0, 5, 0]
C
0
0
1
Entry 4: Cost: 4 Time: 6 X: [2, 0, 6, 2, 6]
0
1
Entry 5: Cost: 5 Time: 6 X: [0, 5, 6, 5, 6]

Figure 4: Algorithm Execution Example

BAS A, and updates the time spent in A and the OR-gate. The left Entry 1 only updates location. Now the priority queue contains two Entries of which Entry 1 is popped first, as it has the least cost. Here, again, two Entries are created: the left transition is discarded as it does not lead to a successful attack, indicated by the tree leaf = 0; the right transition, Entry 3, gains the cost of BAS B and updates the time spent in B, and the OR-gate. If a BAS is executed, we add that time to the vector X. $X[0]$ denotes the time spent for BAS A, 0 if not executed and 2 if executed. When a BAS is executed, we also consider the parent gates, AND/OR/SAND, in the original AT. In the illustrated AT in figure 2, this requires us to take $min(a, b)$ for the OR-gate and $max(or - gate, c)$ for the AND-gate. Thus, if we execute BAS B, the time spent in the OR-gate is $X[3] = 5$. These gates allow us to store the current time spent on the attack, which is stored in the vector X for each node, such that no recalculation of gate times is required.

As we keep taking the least cost Entries from the Priority Queue, we will find solutions with the least cost first. Thus, we find Entry 4 first, but by continuing the Priority Queue's iteration, we can obtain the next K minimum cost solutions. Solution Entries correspond to 1-leaves in the BDD, and they satisfy the original AT.

Algorithm 2: BDD Traversal

```
Input: BDD
Output: Set of Entries that satisfy the AT
begin
    /* Add root Entry with certain ordering */
    $solutions \longleftarrow \emptyset$ ;
    $Queue\ q \longleftarrow \{(time, 0, 0, root, [0, 0, ...0]\}$ ;
    while $q$ not empty do
        Entry E ⟵ q.pop() ;
        Location loc = E.location ;
        if $loc$ = 0-leaf || $loc$ = 1-leaf then
            if $loc$ = 0-leaf then
                continue ;
            else
                $solutions \longleftarrow solutions \cup E$ ;
        else
            BAS ⟵ loc.BAS ;
            /* Left transition entry */
            Entry left ⟵ E ;
            left.location ⟵ loc.left ;
            q.add(left) ;
            /* Right transition entry */
            Entry right ⟵ E ;
            right.location ⟵ loc.right ;
            right.cost ⟵ right.cost + BAS.cost ;
            right.X[BAS.ID] ⟵ BAS.time ;
            right.updateParentTimes(BAS.ID) ;
            q.add(right) ;
    return $solutions$
```

4.1 Attack Values

Attack Values are stored in Entry solutions. Due to the nature of the Priority Queue, it is also easily achieved to get the AT's top-K minimum cost/time attacks. This can be beneficial in the evaluation of larger ATs where the total number of solutions might surpass the threshold of calculating all solutions within a reasonable time.

4.2 Attack Paths

To determine Attack Paths, we traverse the X vector according to the total ordering of BASs and filter out uninvolved BASs to obtain the Attack Path.

4.3 Pareto Curves

Pareto Curves can be determined by continuously iterating over the Priority Queue until it is emptied. We will have all the minimal cut sets (MCSs) in the solution set at this point. These MCSs consist of BASs and the removal of a single BAS causes the MCS solution to be invalid. If $\{a, b\}$ is a MCS, then $\{a, b, c\}$ could be a CS included in the solution set. These CSs can be filtered out during the Pareto Curve construction as we know that costs cannot increase with additional time.

4.4 Ordering

After receiving a particular Entry E solution, it is possible to determine the various other possible orderings. The topological sort can compute these additional ordering by using the sets of BASs, A_x.

Let us define new sets B_x such that $x \in B_x \iff (x \in A_x \ \&\& \ x \in E)$. These m sets of B_x can calculate the total amount of additional solutions in $O(m)$ using the following formula.

$$solutions = \prod_{i=1}^{i=m} |B_i|! \tag{3}$$

From the equation for the number of solutions, it becomes apparent that listing each ordering is often unfeasible due to the factorial complexity of solutions. However, if we index solutions we can query the K'th additional solution in $O(\sum_{i=1}^{i=m} |B_i| * log(|B_i|)^2)$† which scales quasilinearly in the amount of BASs in the solution.

4.5 Limitations

4.5.1 BDD Size

The BDDs formed by total ordering is not always minimal. Between several total orderings, there can be large discrepancies in size. It is essential to reduce the size of the BDD as linear increases in the size of the BDD can lead to an exponential increase on the number of Entries in the algorithm as the number of paths in a BDD are 2^n worst-case where n is the number of nodes in the BDD.

Unfortunately, determining the minimal total ordering is NP-hard [14], but specific heuristics might improve the size of the BDD on the average case.

4.5.2 Cut Sets

The algorithm can provide top-K rankings of attacks based on time, for example, but it could be that 'duplicate' attacks are present in the ranking. An attack could be a superset of another attack. For example, we could have $\{a, b\}$ and $\{a, b, c\}$ as valid CSs in our ranking. However, in this case, the BDD does create MCSs and additional CSs, which might not be useful in the analysis of the AT.

4.5.3 Time Sorting

The Entries are either sorted on time or cost. The algorithm computes these parameters before inserting an Entry into the Priority Queue. Whereas the cost is determined $O(1)$, time is $O(n)$ in the worst-case, which increases the algorithm's complexity.

†To achieve this complexity we use base conversion and a custom list data structure to remove and query items from in logarithmic time. This data structure uses a binary search and lazy segment trees. Further explanation can be found in the Github repository created for this work [15], where all experiments can be replicated.

5. IMPROVEMENTS

5.1 Subsuming

Subsuming was introduced for BDDs [20], which minimizes the size of the BDD, and eliminates 'duplicate' attacks as introduced in section 4.5.2. Using the defined ITE equations, it becomes feasible to apply subsuming throughout the construction of the BDD.

During construction of an $ITE(x, G, H)$, it removes paths from G, which are present in H. In such a case, the solution of H is Y, while the solution in G is $Y \cup x$. Thus, by eliminating the paths in G, we eliminate the superset. The following equation is used for combining ITE structures G and H with variables x and y [20].

$$G \setminus H = \begin{cases} ITE(x, G_1 \setminus H, G_2 \setminus H), & x < y \\ G \setminus H_2, & x > y \\ ITE(x, G_1 \setminus (H_1 or H_2), G_2 \setminus H_2), & x = y \end{cases}$$

The subsuming operation is performed after each ITE construction with a worst-case of $O(n)$. Thus, a factor of n is added to the complexity of the construction. However, in practice, it can still be efficient even on trees with MCSs reaching 10^5 [12].

We wrote a custom implementation for the subsuming operation in Java, as the paper did not provide an implementation. Furthermore, the paper provides an optimization to store results of $G \setminus H$ in a hash table. As the authors did not provide a hash function, the string interpretation of the ITE structure was utilized by the implementation as the key as it is unique for each structure. This hashtable can improve the subsuming operation as results are calculated once for each $G \setminus H$.

5.2 Time Sorting

If minimum cost attacks or Pareto Curves are determined, there is no necessity to track time throughout traversal. In the case the solution Entry has m BASs, the time tracking complexity is $O(n * m)$ worst-case. However, as the Priority Queue does not require time during traversal for minimum cost attacks and Pareto Curves, we can simply ignore time tracking until the end. If it is required, we can calculate the time in the end for a solution in $O(n)$. By removing these superfluous computations, the complexity of minimum cost attacks and Pareto Curves is reduced by a factor m.

6. EVALUATION

For the evaluation, we compare the execution times of the conventional, subsuming, and UPPAAL implementation as proposed in [16] for various ATs from literature [12, 17, 19]. It was essential to take ATs with DAGs AND/OR/SAND-gates. Furthermore, some ATs that benefit from subsuming, while others are already minimal.

6.1 Implementation

We have written the implementations for the conventional and subsuming algorithms in Java, which are available on Github [15]. The UPPAAL algorithm was executed using UPPAAL [7] version 4.1.23.

6.1.1 Conventional & Subsuming

The implementation takes ATs represented in JSON format. The algorithm creates the AT and BDD and uses these to evaluate the algorithm queries. At each step, the AT and BDD structures can be converted to DOT to visualize results and verify correctness. The results are defined in cost, time, and BASs used in the attack. Furthermore, optionally the other orderings of solutions can be computed if required.

The subsuming utilizes the same functions specified by the conventional algorithm. However, it applies the BDD reduction during construction.

6.1.2 UPPAAL

From the AT, UPPAAL files are generated for evaluation. It produces the System Declarations and Declarations files. The library [15] provides a sample UPPAAL file that can be used as a UPPAAL template to evaluate the queries for minimum cost and time. The previous paper [16] can provide supplementary information about the UPPAAL queries and models.

6.2 Experimental Setup

6.2.1 Replications of trees from literature

ATs were gathered from literature, but commonly these were quite small to illustrate specific concepts. To evaluate on larger examples, ATs were combined using AND/OR/SAND-gates. We take two ATs and combine them using an AND/OR/SAND-gate as the root nodes. This can continue multiple replications, allowing us to scale up ATs linearly. Combinations using AND/SAND-gates yield similar results, as it simply changes the ordering of the BDD. Thus, various orderings might influence the computation time between ATs, but the algorithm reaches equal execution times on average.

Equations (4) and (5) give the number of MCS solutions for k replications of the AT, where n denotes the number of MCS solutions of the initial AT.

$$OR = n * k \tag{4}$$

$$AND = n^k \tag{5}$$

These formulae can be used to check that the correct number of solutions was found, and also get the number of solutions for ATs for which we cannot calculate every solution.

6.2.2 Parameters

The ATs are combined using three types of gates, after which the algorithm computes the minimum cost/time and Pareto Curves. Based on these combinations, we can examine the implementations. However, Uppaal does not have a single query to determine the Pareto Curve. Instead, it uses sequential queries that exclude previous results. To account for this, we multiply the execution time of a single query by the amount of MCSs solutions, as this would be the number of required queries.

7. RESULTS

Execution times were measured five times to account for variance, which is indicated by whiskers. However, due to the low variation in results, these whiskers are not always visible. ATs from the literature were used to evaluate the algorithms, of which we highlight two to explain key findings and results.

AT 1: The first AT is for forestalling software, which is comparable to the AT used in the paper [16] except that defense nodes were removed. The created BDD for the AT does not contain additional CSs, which makes the subsuming operation superfluous.

AT 2: The second AT is from [12], and highlights an example where the BDD contains additional CSs. We use these figures to analyze the benefits of the subsuming operation.

Figures 5 and 6 plot the number of MCSs, respectively for 11 and 10 replications of AT 1 and 2. The x-axes show the number of nodes of each AT replication: for AT 1 in figure 5, a single replication contains 19 nodes; the AT representing two replications contains 39 nodes, etc. This is plotted against the total amount of CSs and MCSs in the BDD. While AT 1 contains no additional CSs, AT 2 contains additional CSs, which can be used to determine when the subsuming operation becomes preferred.

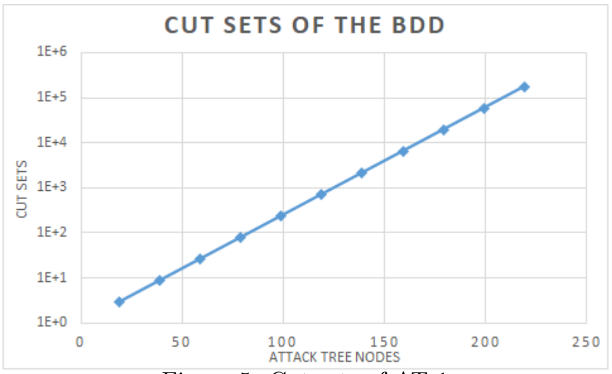


Figure 5: Cut sets of AT 1

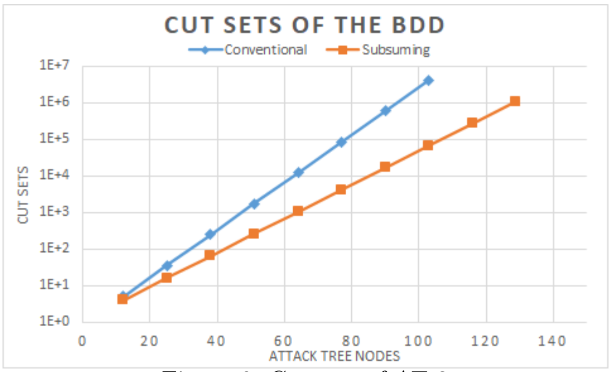


Figure 6: Cut sets of AT 2

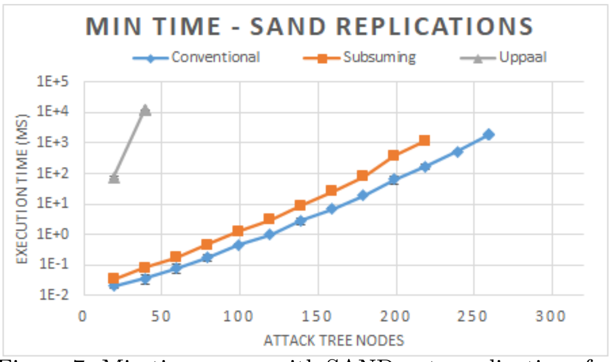


Figure 7: Min time query with SAND-gate replications for AT 1

7.1 Time

In figure 7, we plot the computation runtime of the various algorithms when we query the minimum attack time for AT 1. The x-axis shows the increasing SAND replications of the AT, starting with a single instance (leftmost points of all curves), up to 13 replications (rightmost point of the blue curve). This is plotted against the y-axis, in logscale, which represents the execution runtime, in ms, that it took

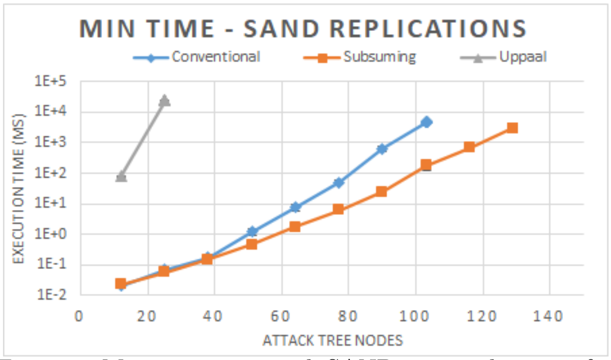


Figure 8: Min time query with SAND-gate replications for AT 2

for each algorithm to compute the solution to the query "what is the time of the fastest attack." Similar plots were created for the minimum cost query and Pareto Curve computation.

In figure 7 and 8, we can observe that the Uppaal algorithm cannot discover solutions within a reasonable time for the third replication of the AT. For AT 1, the subsuming operation is superfluous, which increases the preprocessing time. The traversal of the BDD only occurs until the algorithm finds a single solution, which increases the preprocessing time compared to the traversal. For AT 2, we can observe that the subsuming operation allows us to continue two additional replications due to the reduction in CSs.

For both ATs, we can observe that although the amount of MCSs exceeds 10^6, the minimum time solutions can still be obtained. Thus, obtaining top-K minimum time results can still be viable for larger ATs by using the Priority Queue in the algorithm.

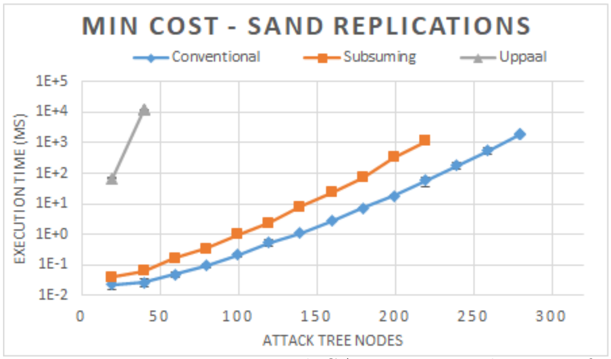


Figure 9: Min cost query with SAND-gate replications for AT 1

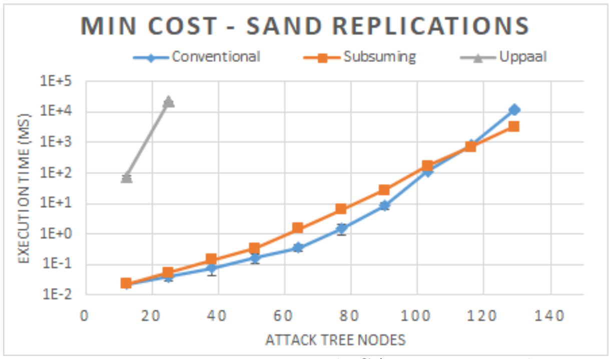


Figure 10: Min cost query with SAND-gate replications for AT 2

7.2 Cost

In figure 9 and 10, we can observe that the Uppaal algorithm cannot discover solutions within a reasonable time for the third replication of the AT. The cost parameter can be tracked with asymptotically lower complexity than time, as discussed in section 4.5.3, which allows us to analyze larger ATs compared to time. However, the execution time does not have to improve significantly. In contrast to Pareto Curves, we only traverse for a single solution, which could take less traversal for time than for cost.

For AT 2, the subsuming operation becomes preferred over the conventional as the CSs and MCSs differ by a factor of 100. Thus, for parameters such as cost and time, the CSs and MCSs have to differ by a significant factor in order to compensate for the increased precomputation.

For both ATs, we observe that the MCSs in the AT surpasses 10^7 and 10^8, but obtaining the least cost solution remains feasible. Thus, similarly to time, obtaining the top-K minimum cost results can still be viable for ATs with increasing amounts of solutions surpassing 10^8 in some cases.

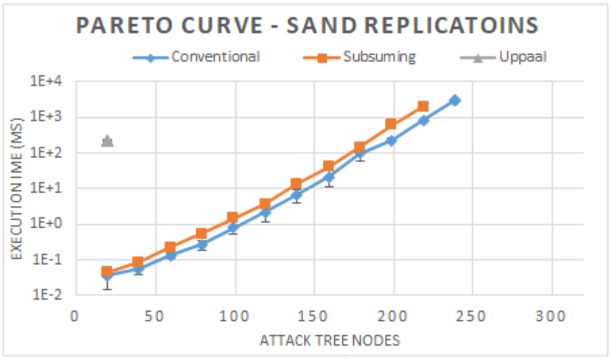


Figure 11: Pareto Curve with SAND-gate replications for AT 1

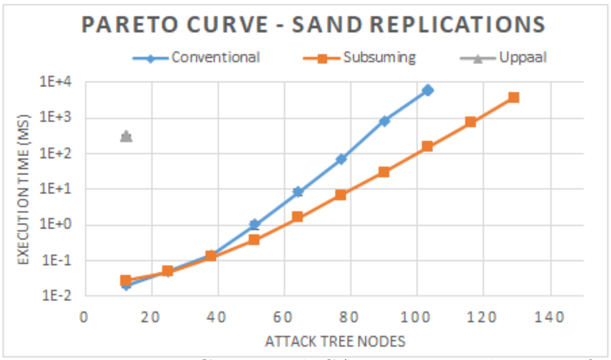


Figure 12: Pareto Curve with SAND-gate replications for AT 2

7.3 Pareto

In figure 11 and 12, we can observe that the Uppaal algorithm cannot discover solutions within a reasonable time for the second replication of the AT. We can observe in AT 1, that for Pareto Curves, the superfluous subsuming operation has less effect on the execution time than single parameters. This is explained by the fact that traversal becomes a more substantial part of the execution than the preprocessing. As Pareto Curves require the traversal of each solution, each algorithm became unviable around 10^6 (M)CSs, while cost and time could continue beyond this.

However, the subsuming algorithm can significantly outperform the conventional algorithm as the BDD minimization gains increasing effect. As the CSs and MCSs differ by a factor of two, the subsuming algorithm is preferred. This allows it to continue multiple replications, but it still cannot surpass replications where the amount of MCSs grows beyond 10^6.

7.4 Asymptotic vs Practical Complexity

From the pseudo-code, we conclude that the Pareto Curves have a worst-case complexity of $O(2^n * n^2)$, with an additional factor of n for the subsuming algorithm. The factor of $O(2^n)$ is the worst-case for the number of solutions, but additionally, the complexity of the BDD construction. For smaller ATs, as figure 13 illustrates, there is a linear relation between CSs and execution time.

However, we should be critical and analyze larger ATs consisting of thousands of nodes to confirm this. Figure 14 illustrates the BDD construction time. We can observe that even for large ATs, the number of solutions remains the limiting factor for the conventional algorithm. However, for the subsuming algorithm, BDD construction becomes the limiting factor as the nodes exceed 400.

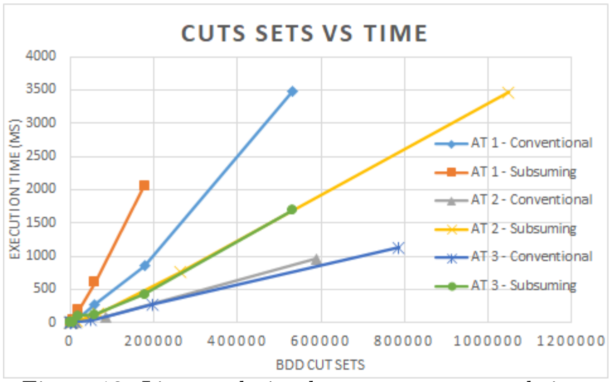


Figure 13: Linear relation between cut sets and time

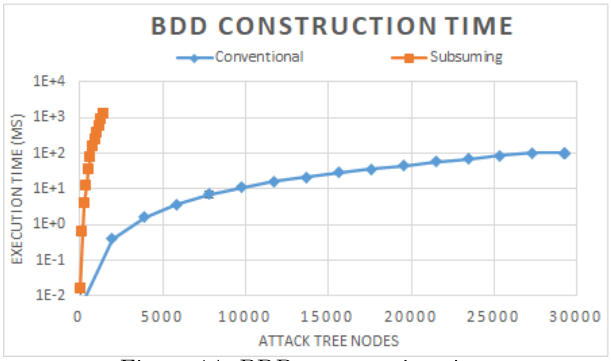


Figure 14: BDD construction time

8. CONCLUSION

The analysis of ATs with DAGs and SAND-gates could be optimized regarding Attack Value, Attack Path, and Pareto Curve metrics. By conversion to BDDs, the issues associated with DAGs could be solved. By imposing a total ordering on the BASs, the BDD data structure was able to abstract the SAND-gates from the AT, as solutions would satisfy the SAND-gates due to the ordering of BASs.

In each of the analyzed ATs in the library [15], the conventional and subsuming algorithms were able to outperform the Uppaal model significantly. From the pseudo-code, we conclude that the cost and Pareto Curve algorithms' complexity have a worst-case of $O(2^n * n)$, where n denotes the number of nodes in the AT. The minimum time query has an additional factor of n during traversal, which leads to a complexity of $O(2^n * n^2)$. The subsuming operation adds a factor n to the complexity of each algorithm.

However, from the results, we can conclude that the algorithm performs much better in practice than the worst-case complexity. The number of solutions is the main limiting factor for each of the algorithms, indicated by the linear relation between solutions and execution time in figure 13. Thus, the metrics could be computed for ATs consisting of thousands of nodes if they have a limited number of solutions. However, for larger ATs, the subsuming operation can significantly impact the construction time, thus preferring the application of the conventional algorithm, as indicated in figure 14.

Attack Values could be computed in asymptotically lower complexities than the state-of-the-art for ATs using the proposed algorithm with optional subsuming for the BDD as an improvement to the conventional algorithm to eliminate non-minimal solutions in the BDD. The subsuming operation for minimum Attack Values, however, takes substantial preprocessing. Thus, it was less efficient than the conventional algorithm, unless there was a considerable discrepancy between the CSs and MCSs as found in AT 2. As the algorithm obtains solutions without traversing every solution, it could still compute minimum parameter solutions for ATs with over 10^8 solutions in some cases.

Attack Paths are computed during the computation of Attack Values. This was expected as Attack Values and Attack Paths are linked quite closely together. Extensions on the topological sort allow us to find additional Attack Paths in quasilinear time with different execution orders. As the amount of these solutions grows with a factorial complexity, an algorithm was developed to query them in quasilinear time.

Pareto Curves are created by traversal of all solutions. From the ATs, we can conclude that the Pareto Curves mainly depend on the amount of CSs generated by the BDD, as each solution has to be traversed. However, as the ATs grow, these complexities can no longer be reached with the subsuming algorithm, but solely by the conventional algorithm.

In future work, the alternative definitions of SAND-gates from literature could be considered, as the framework supports various definitions, as discussed in section 3.2. Furthermore, the probabilities of BASs could be considered for the analysis of ATs to improve the ranking of attacks as the probability of success of a BAS is not currently taken into account.

9. REFERENCES

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Trans. Comput.*, C-27(6):509–516, 1978.

[2] R. Alur and D. Dill. The theory of timed automata. *LNCS*, 600:45–73, 1992.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, apr 1992.

[4] Amenaza. SecurITree. *https://www.amenaza.com/*.

[5] F. Arnold, H. Hermanns, R. Pulungan, and M. Stoelinga. Time-Dependent Analysis of Attacks. *Proceedings of the Third International Conference on POST*, 2014.

[6] Z. Aslanyan, F. Nielson, and D. Parker. Quantitative verification and synthesis of attack-defence scenarios. *Proceedings - IEEE CSF Symposium*, 2016-Augus(Csf), 2016.

[7] G. Behrmann, A. David, and K. G. Larsen. A

Tutorial on UPPAAL. In *LNCS*, number October, pages 200–236, 2004.

[8] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Priced timed automata: Algorithms and applications. In *LNCS*, volume 3657, pages 162–182, 2005.

[9] A. Buldas, P. Laud, J. Priisalu, M. Saarepera, and J. Willemson. Rational choice of security measures via multi-parameter attack trees. *LNCS*, 4347:235–248, 2006.

[10] B. Fila. Efficient attack – defense tree analysis using Pareto attribute domains. *IEEE 32nd Symposium (CSF)*, 2019.

[11] Isograph. AttackTree+. *https://www.isograph.com/software/attacktree/*.

[12] W. S. Jung, S. H. Han, and J. Ha. A fast BDD algorithm for large coherent fault trees analysis. *RELIAB ENG SYST SAFE*, 83(3):369–374, 2004.

[13] A. Jürgenson and J. Willemson. Computing Exact Outcomes of Multi-parameter Attack Trees. In *On the Move to Meaningful Internet Systems: OTM 2008*, pages 1036–1051, 2008.

[14] B. Kaiser and A. Zocher. BDD complexity reduction by Component Fault Trees. *ESREL 2005*, 1(March):1011–1019, 2005.

[15] J. Karsten. Implementation Github. *https://github.com/jarik1999/Fault-Tree-Analysis*, 2020.

[16] R. Kumar, E. Ruijters, and M. Stoelinga. Quantitative Attack Tree Analysis via Priced Timed Automata. *International Conference on FORMATS*, 1:156–171, 2015.

[17] S. Mauw and M. Oostdijk. Foundations of Attack Trees. *LNCS*, 3935:186–198, 2006.

[18] D. J. Pearce and P. H. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM JEA*, 11(1):1–24, 2006.

[19] L. Pietre-Cambacedes and M. Bouissou. Attack and defense modeling with BDMP. *LNCS*, 6258:86–101, 2010.

[20] A. Rauzy. New algorithms for fault trees analysis. *RELIAB ENG SYST SAFE*, 40(3):203–211, 1993.

[21] B. Schneier. Attack Trees - Modelling Security Threats. *Dr. Dobb's Journal*, 1999.

[22] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated Generation and Analysis of Attack Graphs. *Kaos GL Dergisi*, 26(2):147–173, 2002.

[23] J. Weiss. A System Security Engineering Process. *14th National Computer Security Conference*, pages 1–26, 1954.

APPENDIX

A. SEQUENTIAL ATTACKS

In the case of sequential attacks, the computation of time for Entries is simplified. Similarly to cost, a time variable can be stored, to which time of a BAS is added. The vector X can be simplified to a Boolean array to store which BASs were involved in the solution. The complexity for minimum time reduces by a factor of n, as computing time no longer has a worst-case of $O(n)$. The complexities of the other algorithms remain the same.

We can note that the additional orderings of BASs for solutions are more relevant to sequential attacks as (a, b, c) and (b, a, c) have a distinct ordering of BASs in sequential attacks. In contrast, for parallel attacks, a and b are executed in parallel in both cases.