Most Vulnerable Attack Trace in a Probabilistic Attack Graph

Xuanli Lin, Zhaofeng Zhang, Yinxin Wan, Ethan Teo, Guoliang Xue, Yanchao Zhang

***Abstract*—In this paper, we investigate the properties and computation of attack traces on probabilistic attack graphs, a model that integrates probability into traditional attack graphs to reflect the varying exploitability of network vulnerabilities. We introduce the Most Vulnerable Attack Trace (MVAT) problem, which aims to identify the attack trace with the highest cumulative success probability for an attacker. To address this problem, we propose both an exact algorithm and a heuristic algorithm, each designed to navigate the complexities introduced by cycles in the attack graph. Our exact algorithm explores all possible sequences of node selections to ensure the optimal attack trace, while our heuristic algorithm efficiently approximates the MVAT in polynomial time. We evaluate the performance of our algorithms using an extensive dataset, demonstrating the usefulness of the proposed algorithms.**

1. Introduction and Background

Cybersecurity has garnered significant attention in the research community, and recent high-profile cybersecurity incidents [2] have highlighted the importance of understanding and mitigating sophisticated cyber threats. The proliferation of interconnected systems and the increasing complexity of network architectures have only expanded the landscape for potential cyber-attacks. For example, the rise of Internet of Things (IoT) has brought smart devices to the masses, but vulnerabilities in these devices have also introduced a wide range of attacks [1]. Therefore, a systematic view of the networks and how attackers may penetrate them is vital to understanding network security.

To this end, *logical attack graph* [7] was proposed and has been used in a wide variety of network vulnerability studies [9] [10]. The attack graph visualizes the vulnerabilities of connected devices, where each node represents a fact or rule, and each edge represents a "depends on" relationship.

One emerging area in this domain is the study of the *attack trace* in attack graphs, proposed in IOTA [3]. An attack trace is a subgraph of an attack graph, and it models how an attacker can achieve their attack goal from existing knowledge and vulnerabilities, and provides an important insight into attacker's possible course of action. The attack trace represents a structured way to represent the sequence of actions an attacker can take, illustrating the interplay between different vulnerabilities and the progression of an attack. Specifically, IOTA performs an in-depth analysis of common IoT devices to extract information such as their vulnerabilities and interactions with the physical world. This information is then used to construct an attack graph, and it finds the shortest attack trace (SAT) in the graph, defined as an attack trace with the shortest depth. Here, the depth is the hop count of the longest path from any source node to the sink node in the attack trace.

Wan et al. [11] augments the works in IOTA and proposed a variant of the attack graph model, where nodes and edges are assigned non-negative *weights*. This weight can represent concepts such as the penetrability of a node or an edge. Accordingly, an SAT is defined as an attack trace with the minimum height in the graph, where the height is calculated as the sum of the weights of nodes and edges along the longest path within the attack trace.

We aim to further refine the attack trace model by introducing the notion of probability. The concept of probability in attack graphs and its application in attack graphs is well discussed in the literature. For example Wang et al. [12] assign probabilities to each node in an attack graph and calculate the cumulative probabilities for every node in the attack graph. Munoz-Gonzalez et al. [5] proposes a Bayesian approximation scheme for acyclic attack graphs where each node is a random variable and each edge represents a dependency relationship between two variables. However, to the best of our knowledge, the attack trace has not been used in a cyclic probabilistic attack graph. We study the attack trace in a variant of logical attack graphs with probabilities at nodes and/or edges, and we define the *most vulnerable attack trace* (MVAT) in attack graphs. We examine how cycles in an attack graph can complicate the derivation of MVAT. Then, we propose an exact algorithm and a heuristic algorithm to solve MVAT in any attack graph, respectively. Our experiment results show that both algorithms can produce better results compared to the baseline, a modified SAT algorithm. Our main contributions are the following:

- We study the properties of the attack trace in probabilistic attack graphs and formulate the MVAT problem.
- We propose an exact algorithm and a heuristic algorithm for solving the MVAT problem, which correctly determines the MVAT in any attack graph.
- We implement both algorithms and perform evaluation on an extensive dataset. The results demonstrate the effectiveness of our algorithms.

The remainder of this paper is organized as follows. Section 2

Xuanli Lin, Zhaofeng Zhang, Ethan Teo, Guoliang Xue, and Yanchao Zhang are affiliated with Arizona State University and CoE-FutureG. Email: {xlin54, zzhan199, ethanteo, xue, yczhang}@asu.edu. Yinxin Wan is affiliated with UMass Boston. Email: Yinxin.Wan@umb.edu.
Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-23-2-0225. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

defines basic concepts. Section 3 discusses some properties of the attack trace in probabilistic attack graphs. Section 4 and 5 presents our exact and heuristic algorithm for computing an MVAT, respectively. Section 6 presents the evaluation results. Section 7 concludes this paper.

2. BASIC CONCEPTS

We extend the concept of attack graphs [7] and introduce probabilistic attack graphs.

Definition 1 (Probabilistic Attack Graph). A probabilistic attack graph is a directed graph $G = (V_p, V_d, V_r, E, p, g)$, where V_p, V_d and V_r denote the set of *primitive fact* vertices (`source` vertices), *derived fact* vertices (`OR` vertices), *rule* vertices (`AND` vertices), respectively (we use $V = V_p \cup V_d \cup V_r$ to denote the set of all vertices); $E \subseteq \{(V_p \cup V_d) \times V_r\} \cup \{V_r \times V_d\}$ is the set of directed edges; $p(v) : V \to \mathbb{R}_{\geq 0}$ is the *success probability* for a vertex $v \in V$; $p(e) : E \to \mathbb{R}_{\geq 0}$: the *success probability* for an edge $e \in E$; $g \in V_d$ is the attacker's goal or the sink vertex.

Here, the *success probability* refers to the probability that an attacker successfully attacks and gains access to a network element if the preconditions on that element is satisfied.

We study the type of attack graph where the attacker has a single goal g; however the work can be trivially extended to the case where multiple goals exist. □

The concept of *probabilistic attack graph* is based on the concept of *attack graph* studied in the literature [3, 7, 9, 10], where the success probability p is applied to the edges and/or vertices of a regular attack graph. Unless otherwise specified, an "attack graph" refers to a probabilistic attack graph for the rest of the paper.

Since vulnerabilities differ greatly in their exploitability, the success probability is a natural expression of the overall difficulty for an attacker to exploit the network element.

To analyze the vulnerabilities in enterprise networks, the concept of attack path was used in [7], and it is an acyclic sequence of vertices $< v_p, v_d^1, v_d^2 \ldots, g >$ where v_p is a leaf vertex and v_d^1, v_d^2, ... are derived vertices. It was demonstrated that the concept of attack path does not fully capture the dependencies between vertices in an attack graph [11]. Therefore, our work relies on the concept of attack trace proposed in [3]. We define the attack trace in a probabilistic attack graph as follows.

Definition 2 (Attack Trace in a Probabilistic Attack Graph). Let G be a probabilistic attack graph. Let $t \in V$ be any vertex in G. An *attack trace to vertex* t in G, denoted by T^t, is a subgraph of G that satisfies the following properties:

1) Let v be any vertex in T^t. If $v \in V_d$, then the in-degree of v in T^t is at least 1. In other words, for any `OR` vertex v in T^t, T^t contains at least one of the edges in $\{(u, v) | (u, v) \in E\}$.
2) Let v be any vertex in T^t. If $v \in V_r$, then the in-degree of v in T^t equals to the in-degree of v in G. In other words, for any `AND` vertex v in T^t, T^t contains all edges in the set $\{(u, v) | (u, v) \in E\}$.
3) Let v be any vertex in T^t. If the in-degree of v in T^t is 0, then $v \in V_p^G$. In other words, every source vertex in T^t is a primitive fact vertex.
4) Vertex t is the only vertex in T^t with out-degree 0 in T^t.
5) T^t is acyclic.

An attack trace to vertex g is called an attack trace of G. We may use T to denote T^g since the goal vertex g is unique in G. □

To obtain the probability that the goal node is successfully attacked, we introduce the cumulative success probability for each node in an attack graph.

Definition 3 (Cumulative Success Probability). Let G be a probabilistic attack graph. Let $t \in V$ be any vertex in G. Let T^t be an attack trace to t in G.

Let v be any node in T^t, the cumulative success probability at v is $P(v)$, and it is recursively defined as follows.

- If $v \in V_p$, $P(v) = p(v)$.
- If $v \in V_r$, $P(v) = p(v) \cdot \prod_{(u,v) \in E(T^t)} P(u) \cdot p(u, v)$.
- If $v \in V_d$, $P(v) = p(v) \cdot (1 - \prod_{(u,v) \in E(T^t)} (1 - P(u) \cdot p(u, v)))$.

We define the cumulative success probability of T^t, denoted $P(T^t)$, as the cumulative success probability at t, i.e., $P(T^t) = P(t)$. □

To identify the attack trace that yields the highest success probability to the goal node (and therefore, most vulnerable), we define the MVAT as follows.

Definition 4 (Most Vulnerable Attack Trace). Let G be a probabilistic attack graph G, and $t \in V$ be any vertex in G. Let T_{mv}^t be an attack trace in G. T_{mv}^t is said to be a MVAT to vertex t in G, if $P(T_{\text{mv}}^t) \geq P(T^t)$ for any attack trace T^t to vertex t. A MVAT to vertex $g \in V$ (the attacker's goal) is called a MVAT of G. □

A given attack graph may have multiple nonidentical MVAT. However, for any two MVATs T_1 and T_2 of attack graph G, their cumulative success probability must be equal, i.e., $P(T_1) = P(T_2)$. For ease of presentation, we use T_{mv}^G to denote a MVAT; we drop the superscript and say T_{mv} when the referred graph G is clear from the context.

Following a MVAT is the *optimal* strategy for the attacker. In other words, for an attacker, following the MVAT T_{mv} would represent the best likelihood of them gaining access to the goal node.

3. PROPERTIES OF AN ATTACK TRACE

Given a (probabilistic) attack graph G, we observe the following properties about its attack trace T^t:

Proposition 3.1. *For any AND node $v_r \in V_r$, if it is included in T^t, then all its predecessors and the edges pointing from these predecessors to v should also appear in T^t.*

Proof. This follows directly from Definition 2. □

Proposition 3.2. *If any OR node $v_d \in V_d$ is included in T^t, having more of its predecessors (and their associated edges)*

always yields equal or better cumulative success probability $P(v_d)$ at the node.

Proof. This result stems from our Definition 3, where success probability at OR nodes are multiplicatively computed.

Consider the following example, where an OR node v_d with two predecessors u_1 and u_2. When no predecessor is selected in T^t, the probability of reaching v_d is 0. (In fact, v_d would not be included in T^t as it would be unreachable). When only u_1 is selected, $P(v_d)_1$ can be given as

$$P(v_d)_1 = P(u_1) \cdot p(u_1, v_d) \cdot p(v_d) \tag{1}$$

When both u_1 and u_2 are selected, the probability becomes

$$\begin{aligned} P(v_d)_2 = (1-(1 - P(u_1) \cdot p(u_1, v_d)) \cdot \\ (1 - P(u_2) \cdot p(u_2, v_d))) \cdot p(v_d) \end{aligned} \tag{2}$$

Subtracting equation (1) from equation (2), their difference is

$$\begin{aligned} P(v_d)_2 - P(v_d)_1 = P(u_2) \cdot p(u_2, v_d) \cdot p(v_d) \cdot \\ (1 - P(u_1) \cdot p(u_1, v_d)) \end{aligned} \tag{3}$$

Since the node and edge probabilities are always non-negative, quantity (3) is also non-negative. □

An interesting result of the above propositions is given below

Corollary 3.1. *For an acyclic probabilistic attack graph G, its MVAT T_{mv} is the attack graph itself, i.e., $T_{\text{mv}} = G$.*

Proof. Assume the opposite is true: an MVAT of G can be obtained by removing nodes from the original graph G. Suppose v is one of the removed nodes. If $v \in \{V_p \cup V_d\}$, it would also remove its successor AND nodes to satisfy the definition of an attack trace, which is equivalent to the following situation. If $v \in V_r$, its successor OR nodes will have one less predecessor. As shown in Proposition 3.2, this will result in equal or worse cumulative probabilities of successor nodes of v. Therefore, MVAT cannot be obtained by removing nodes from the original attack graph. □

In this paper, we are mainly interested in cyclic attack graphs. If we select the entire graph as an attack trace, there will be cycle(s) in the attack trace, which is not permitted. Therefore, we must consider the properties of a cyclic attack graph and how we can produce a cycle-free attack trace while maximizing the success probability at the goal node.

Formally, we study the following problem.

Definition 5 (MVAT Problem). *Given a probabilistic attack graph G, what is a MVAT, T_{mv} in G? And what is the corresponding cumulative success probability at the goal node, $P(T_{\text{mv}})$?*

As discussed in Corollary 3.1, the answer to the MVAT Problem in an acyclic attack graph is trivial. For cyclic graphs, the exact answer is not easily obtained.

Consider the following high-level algorithm for obtaining an attack trace T^g in attack graph G (some lesser details omitted). First, create a priority queue PQ. Then, all the primitive nodes V_p to PQ, and update $P(v)$ for all $v \in V_p$. While the PQ not empty, pop u from PQ, then visit u's neighbors v: update $P(v)$ and if v is ready, insert v to PQ and insert (u, v) to T^g. Finally, return T^g and $P(T^g)$ when g is extracted from PQ.

It is easy to determine whether a node is considered ready for an AND node - it is ready only when it is visited by all of its predecessors. However, for an OR node, the answer is more nuanced if we want to obtain an MVAT from G.

As stated in Proposition 3.2, for an OR node v, it is advantageous to include as many of its predecessors as possible in the attack graph to maximize $P(v)$. However, if there are cycle(s) that v is part of, we will have to decide when we consider v to be ready. A natural approach is to wait for every other node to be visited and processed and only add v to PQ. We can then get the maximum possible $P(v)$.

However, if there is a deadlock, i.e., multiple OR nodes are not ready after we exhaust the PQ, it may indicate that some OR nodes are the predecessor of some other OR nodes. In this circumstance, it is necessary to resolve the deadlock by picking an OR node (that is not ready when the PQ is exhausted) so the algorithm can proceed. Eventually, there can be another deadlock that needs to be resolved, and so on. The process continues until we find an attack trace when the goal g is extracted from the PQ, or we fail to find an attack trace when all nodes $V \in G$ are visited, and g is not extracted.

Therefore, the crux of the MVAT Problem is the determination of an optimal sequence of OR nodes to be inserted into the PQ when deadlocks happen. To this end, we propose an optimal exact algorithm in Section 4 and a heuristic in Section 5. The difference between the two algorithms lies entirely in their deadlock resolution strategies. While the heuristic algorithm resolves the deadlock by picking the non-ready OR node with the highest cumulative success probability, the exact algorithm attempts all possible sequences of OR node selection to find the one that results in the highest cumulative success probability at the goal node.

4. Exact Algorithm for MVAT

We present an optimal algorithm `ExactMVAT` for computing an MVAT in Algorithm 1. The algorithm's flow is as follows:

We first initialize the attack graph G in Line 1 to Line 10. Then, we visit all primitive fact nodes V_p and update their cumulative success probability in Line 11, using the `Advance` auxiliary procedure.

Essentially, `Advance` takes in an attack graph G and a queue Q and attempts to visit every successor node of the nodes in Q. A node is only visited if its predecessor nodes have been visited. Ultimately, it may end up with zero or more nodes that cannot be visited due to their predecessors not being fully visited and no other nodes being available. The algorithm then returns a list L of OR nodes in deadlock.

L may be empty, indicating that no deadlock has been encountered. If there are multiple nodes in L, we need to try all of them to find out which one gives the most optimal cumulative success probability at the goal node. The process effectively results in a decision tree if multiple such decisions are needed.

In Algorithm 1, we use a stack S and an indicator i to guide the algorithm's execution. When a deadlock is encountered, a stack entry (L, G, i) is push onto the stack S (Line 16). The indicator i is initially set to 1, and the algorithm proceeds to call `Advance` on G, with Q being just a single node $L[i]$ (Line 17). This has the effect of choosing $L[i]$ to break the deadlock.

After advancing, if there are no nodes left in the resulting L, we know that either an attack trace is found or no attack trace exists. In Line 19 and Line 20, we update the global best and its corresponding attack trace. Note that if no attack trace is found, $P(g)$ is zero, so no update will occur.

In Line 21 to Line 23, we check if There are entries on the stack. If so, we will backtrack and pop the top entry. The algorithm then increments the counter i and proceeds to `Advance` on $L[i]$. Otherwise, there are no more tree nodes left to explore on the decision tree, and we can go to Line 25.

If L is not empty, we need to explore the nodes in L, and we create a new layer of stack entry.

In conclusion, `ExactMVAT` traverses the decision tree for breaking deadlocks in a DFS manner and finding a sequence that maximizes the cumulative success probability at g.

Algorithm 1: ExactMVAT(G)

Input: $G = (V_p, V_d, V_r, E, p, g)$: an attack graph
Output: T_{mv}: an attack trace to g in G with the highest probability of success; $P_{\max}$: success probability of g in T_{mv}

1 Create an empty queue Q; Create an empty list L; Create an empty stack S;
2 $P_{\max} \leftarrow 0$; $T_{\text{mv}} \leftarrow nil$;
3 **for** $\forall v \in V_d$ **do**
4 $v.\text{in} \leftarrow 0$; $v.\text{done} \leftarrow 0$; $P(v) \leftarrow 0$; $v.\text{color} \leftarrow$ `WHITE`;
5 **for** $\forall v \in V_r$ **do**
6 $v.\text{in} \leftarrow 0$; $v.\text{done} \leftarrow 0$; $P(v) \leftarrow 1$; $v.\text{color} \leftarrow$ `WHITE`;
7 **for** $\forall e = (x, y) \in E$ **do**
8 $y.\text{in} \leftarrow y.\text{in} + 1$;
9 **for** $\forall v \in V_p$ **do**
10 $P(v) \leftarrow p(v)$; Insert v to Q; $v.\text{color} \leftarrow$ `GRAY`;
11 $L \leftarrow$ `Advance`(G, Q);
12 **if** L is empty **then**
13 $P_{\max} \leftarrow P(g)$; $T_{\max} \leftarrow$ construct T; **goto** 25;
14 $i \leftarrow 0$;
15 $i \leftarrow i + 1$;
16 Push (L, G, i) to S;
17 $L \leftarrow$ `Advance`$(G, \{L[i]\})$;
18 **if** L is empty **then**
19 **if** $P(g) > P_{max}$ **then**
20 $P_{max} \leftarrow P(g)$; $T_{max} \leftarrow$ construct T;
21 **if** S is not empty **then**
22 Pop L, G, i from S; **goto** 15;
23 **else goto** 25;
24 **else goto** 14;
25 **output** $T_{\text{mv}}, P_{\max}$;

Theorem 1. *Given an attack graph with n nodes, n_d OR nodes, and m edges. In the worst-case scenario, Algorithm 2 finishes in $O(n+m)$ time, and Algorithm 1 finishes in $O((n+m) \cdot n_d!)$ time.*

Proof. Algorithm 2 spends $O(1)$ time on at most n nodes and m edges. This is because a coloring system keeps track of the visit state of a vertex, and it will never visit the same vertex twice. The overall time complexity is $O(n + m)$.

Algorithm 1 visits every node in a decision tree, in which there are at most n_d nodes at the first level, $n_d \cdot (n_d - 1)$ nodes at the second level, and so on. There are at most n_d such levels, resulting in $n_d!$ leaf nodes in the tree. Each leaf node corresponds to a particular sequence of OR nodes are chosen to break the deadlocks, and for that sequence only, the algorithm visits at most n nodes and m edges. Thus, the algorithm has an overall time complexity of $O((n + m) \cdot n_d!)$. □

Remarks. The running time established in Theorem 1 is not tight, and the actual time complexity varies greatly depending on the graph layout. Specifically, the number of OR nodes held in deadlocks determines the width of the decision tree, while the number of deadlocks determines the tree's height. The theoretical upper bound only occurs when every OR node in a graph is part of a deadlock, and they all depend on each other. In a realistic setting, it is exceedingly unlikely that a large portion of OR nodes are involved in deadlocks, and thus, the running time of the exact algorithm would be acceptable. Evaluation results in 6-B confirms this observation.

Algorithm 2: Advance(G, RQ)

Input: $G = (V_p, V_d, V_r, E, p, g)$: an attack graph, RQ: queue of nodes that were already visited
Output: L: a list of OR nodes that are in a deadlock

1 Create an empty list L;
2 **while** RQ is not empty **do**
3 Dequeue u from RQ; $u.\text{color} \leftarrow$ `BLACK`;
4 **for** $\forall v \in u.\text{adj}$ with $v.\text{color} \neq$ *BLACK* **do**
5 $temp \leftarrow P(u) \cdot p(u, v)$;
6 **if** $(v \in V_d)$ **then**
7 **if** $(v.\text{done} = 0)$ **then**
8 $P(v) \leftarrow temp \cdot p(v)$; $v.\text{fail} \leftarrow 1 - temp$;
9 $v.\text{color} \leftarrow$ `GRAY`;
10 Insert u to $v.\text{parent}$; Insert v to L;
11 **else**
12 $P(v) \leftarrow (1 - v.\text{fail} \cdot (1 - temp)) \cdot p(v)$;
13 $v.\text{fail} \leftarrow v.\text{fail} \cdot (1 - temp)$;
14 Insert u to $v.\text{parent}$;
15 $v.\text{done} \leftarrow v.\text{done} + 1$;
16 **if** $v.\text{done} = v.\text{in}$ **then**
17 Remove v from L; Enqueue v to RQ;
18 **if** $(v \in V_r)$ **then**
19 $P(v) \leftarrow temp \cdot P(v)$; $v.\text{done} \leftarrow v.\text{done} + 1$;
20 **if** $v.\text{done} = v.\text{in}$ **then**
21 $P(v) \leftarrow P(v) * p(v)$; $v.\text{color} \leftarrow$ `GRAY`;
22 Enqueue v to RQ;
23 **output** L;

5. A Heuristic for MVAT

In addition to the `ExactMVAT` algorithm, we design a polynomial-time heuristic algorithm that solves the MVAT

Problem. We observe from Proposition 3.2 that adding a predecessor node to an OR node v_d has a diminishing marginal return for $P(v_d)$ due to the multiplicative nature of the probability calculation. Therefore, a simple strategy for breaking the deadlock can be choosing the OR node with the highest cumulative success probability.

`QuickMVAT` uses this observation to obtain an approximate for the MVAT Problem. Similar to `MVAT`, `QuickMVAT` visits all primitive fact nodes V_p and use `Advance` to proceed as far as it can. When a deadlock is encountered, it picks the node (that is in the deadlock) with the highest cumulative success probability, and resume the process. This process repeats until the goal node is visited or when it runs out of nodes before reaching the goal node.

Algorithm 3: QuickMVAT(G)

Input: $G = (V_p, V_d, V_r, E, p, g)$: an attack graph
Output: T_{hu}: an attack trace to g in G; P_{hu}: cumulative success probability of g in T_{hu}

1 Create an empty queue RQ; Create an empty max priority queue PQ;
2 $P_{\text{hu}} \leftarrow 0$; $T_{\text{hu}} \leftarrow nil$;
3 **for** $\forall v \in V_d$ **do**
4 $\quad v.\text{in} \leftarrow 0$; $v.\text{done} \leftarrow 0$; $P(v) \leftarrow 0$; $v.\text{color} \leftarrow$ `WHITE`;
5 **for** $\forall v \in V_r$ **do**
6 $\quad v.\text{in} \leftarrow 0$; $v.\text{done} \leftarrow 0$; $P(v) \leftarrow 1$; $v.\text{color} \leftarrow$ `WHITE`;
7 **for** $\forall e = (x, y) \in E$ **do**
8 $\quad y.\text{in} \leftarrow y.\text{in} + 1$;
9 **for** $\forall v \in V_p$ **do**
10 $\quad P(v) \leftarrow p(v)$; Enqueue v to RQ; $v.\text{color} \leftarrow$ `GRAY`;
11 **while** RQ is not empty **do**
12 $\quad L \leftarrow$ `Advance(G, RQ)`;
13 $\quad$ Add all nodes $v \in L$ to PQ, weighted by $P(v)$;
14 $\quad$ **if** PQ is not empty **then**
15 $\quad\quad$ Extract u from PQ; Insert u to RQ;
16 construct T_{hu}
17 **output** $T_{\text{hu}}, P(g)$

Theorem 2. *Given an attack graph with n nodes, n_d OR nodes and m edges, Algorithm 3 finishes in $O(n + m + {n_d}^2)$ time.*

Proof. The algorithm spends $O(n + m)$ time to initialize the nodes in Lines 1 to 10. Even though Line 11 can be executed for at most n_d times; the algorithm visits no more than n nodes and m edges in total. This is because the heuristic is a greedy algorithm that does not backtrack, and its selection of OR nodes at deadlocks is final. Therefore, Line 12 contributes $O(n + m)$ total regardless of the graph layout. Line 13 takes $O(n_d)$ every loop iteration, and Line 15 takes $O(\log n_d)$, so the entire `while` loop takes $O({n_d}^2)$ to finish. Thus, the overall complexity of the Algorithm 3 is $O(n + m + {n_d}^2)$. □

6. Performance Evaluation

We implemented both algorithms to evaluate our proposed exact and heuristic MVAT algorithms and tested them in generated attack graphs with randomized probabilities. We introduce the topology and the probability derivation in Section 6-A and present the evaluation results in Section 6-B.

A. Evaluation Setup

We evaluated the algorithms on generated attack graphs with different sizes and edge densities. The graphs exhibit a similar topology to those generated by Mulval in the seminal paper [8]. Still, they can have an arbitrary number of nodes and edges and ensure the presence (or absence) of cycles in the graph. This way, we can explore the behaviors of the two algorithms easily.

The generation process is mainly controlled by the following parameters: n_p, n_d, n_r, and e for the number of primitive fact nodes, derived fact nodes, rule nodes, and edges. First, a graph with a specified number of nodes (of three kinds) is populated, yielding a set of nodes $V = \{V_p \cup V_d \cup V_r\}$. To ensure that the generated result contains at least one attack trace, we track a set of reachable nodes (from the primitive fact nodes), V_{re}, and it is initialized to V_p. The idea is that a node is more likely to be part of an attack trace in the final attack graph if a node is reachable.

Then, the program produces a set of edges E for the graph. It generates a starter attack graph by randomly picking nodes $v_p \in V_p$ and $v_d \in V_d$. Two edges are added to the graph: (v_p, v_r) and (v_r, g), where $g \in V_d$ is the goal node. This graph is guaranteed to have an attack trace (v_p, v_r, g), and we can build an attack graph from this starter graph. v_r and g is added to V_{re}

The remainder of the edge generation process goes as follows. First, the program randomly selects a start node $u \in V_{\text{re}}$. Depending on the type of v_s, it randomly picks an appropriate end node v. Specifically, in case $u \in V_p$ or $u \in V_d$, then $v \in V_r$. If $u \in V_r$, then $v \in V_d$. Before committing edge (u, v) to the graph, the program checks to ensure that

1) $(u, v) \notin E$,
2) the graph after adding (u, v) contains an attack trace,
3) if $u \in V_r$, then $(u, v') \notin E, \forall v' \in V_d$, and
4) optionally, the graph after adding (u, v) does not contain a cycle.

The second criterion can be efficiently checked with the `SAT` algorithm [11], and the third criterion is due to Mulval [8] generates attack graphs with exactly one outgoing edge from each rule node. The fourth criterion is only enforced if no cycle is desired in the graph (and can be checked with, e.g., Kahn's algorithm [4]). If (u, v) satisfies these criteria, it is added to the graph, with v added to V_{re}. Otherwise, another pair of nodes u, v is selected as described. The algorithms do not assume these restrictions and can work on any attack graph.

The program stops when the requested edge count e is met or fails to find a suitable edge after many attempts. Finally, the graph is cleaned to remove unreachable nodes and their edges.

After generating an attack graph, we assign success probabilities on all edges. The probability can be derived from vulnerability scores like [6]. Since the graph is randomly generated for our experiment, we draw the probabilities from a normal distribution $N(0.8, 0.2)$ for evaluation. Our algorithms

do not assume any distribution and work for any probability assignment. All the nodes have a success probability of 1.

For this experiment, we generated graphs with 7 sizes, each with the same number of n_p, n_d, and n_r. The sizes are 5, 10, 20, 40, 80, 160, and 320. Following the third criterion above, it is calculated that an attack graph has at most $n_{\max} = (n_p + n_d) \cdot n_r$ edges. We generated graphs with different densities d for each graph size, where $d = e/n_{\max}$. The densities are 0.1, 0.2, 0.3, 0.4, and 0.5. We generated 100 graphs with different random seeds for each size and density combination.

The baseline of the experiment is an adapted version of the `SAT` algorithm [11]. The difference is that instead of using the height of a node as the key in the priority queue, it now uses the cumulative success probability as the key. We refer the interested reader to the [11] for details.

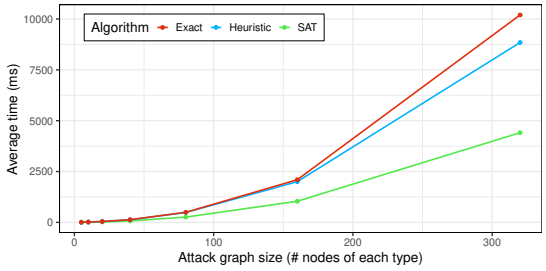
Because of the randomized nature of the graph, directly comparing the cumulative success probability from different algorithms at the goal node does not make sense. Instead, we introduce φ', the ratio between $P(g)$ achieved by our proposed algorithms and a baseline. This ratio φ' is then averaged over the 100 test cases to get the mean φ for each size and density combination.

B. Evaluation Results and Analysis

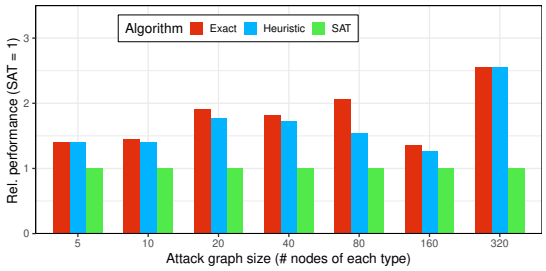
We present evaluation results. Fig. 1(a) depicts the trend in average execution time of three algorithms running the test cases as the graph size increases. Note that the average is taken over a set of 100 test cases and 5 density levels, resulting in 500 observations. The x axis is true to the scale to reflect the trend better. Fig. 1(b) shows the φ of the three algorithms in different graph sizes (averaged over 5 densities).

Fig. 1(a) reveals the growth of running time for the `exact` is close to the `heuristic`, while both of them run above `SAT`. This might be surprising given the theoretical worst-case complexity of `exact` is factorial, while the worst-case complexity of `heuristic` is polynomial. However, as discussed in the remark of Theorem 1, the actual runtime of the `exact` algorithm depends on the number of OR nodes stuck in deadlocks and their connection. Thus, in an average generated case, the topology does not cause a significant divergence in running time of the `exact` and the `heuristic` algorithm.

Fig. 1(b) shows the heuristic algorithm performs satisfactorily compared to the exact algorithm. On average, its φ reaches approximately (percentage) of φ of `exact` in graph size of 5, and about (percentage) in graph size of 80. Both the `exact` and the `heuristic` outperforms the `SAT` due to the latter adding exactly one predecessor of an OR node to the attack trace. It is worth noting that bigger attack graphs are not supersets of smaller attack graphs, therefore the relative ratio of the performace in each size category can vary.



(a) Average runtime in graphs of different sizes



(b) φ in graphs of different sizes

Fig. 1. Evaluation results for the two algorithms.

7. Conclusion

In this paper, we study and analyze the attack trace in a probabilistic attack graph. We define the MVAT problem, and we present several characteristics of an attack trace in an attack graph. We design an exact algorithm and a heuristic algorithm to solve the MVAT problem. Our proposed solutions can be applied to a wide range of problems, including the analysis and hardening of and IoT network. Our experiment shows that both our algorithms can correctly calculate an attack trace in a variety of graphs, especially the ones that have cycles. The heuristic algorithm is able to obtain good approximations of the MVAT in polynomial time.

References

[1] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: Security evaluation of home-based IoT deployments," in *Proc. of IEEE S&P*, 2019.

[2] CSIS, https://csis.org/programs/strategic-technologies-program/significant-cyber-incidents.

[3] Z. Fang, H. Fu, T. Gu, P. Hu, J. Song, T. Jaeger, and P. Mohapatra, "IOTA: A framework for analyzing system-level security of IoTs," in *Proc. of ACM/IEEE IoTDI*, 2022.

[4] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[5] L. Munoz-Gonzalez, D. Sgandurra, A. Paudice, and E. C. Lupu, "Efficient attack graph analysis through approximate inference," *ACM Transactions on Privacy and Security*, vol. 20, no. 3, pp. 1–30, 2017.

[6] NVD, https://nvd.nist.gov/vuln-metrics/cvss.

[7] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proc. of ACM CCS*, 2006.

[8] X. Ou, S. Govindavajhala, A. W. Appel *et al.*, "Mulval: A logic-based network security analyzer." in *Proc. of USENIX Security Symposium*, 2005.

[9] A. Sabur, A. Chowdhary, D. Huang, and A. Alshamrani, "Toward scalable graph-based security analysis for cloud networks," *Computer Networks*, vol. 206, p. 108795, 2022.

[10] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *Proc. of IEEE S&P*, 2002.

[11] Y. Wan, X. Lin, A. Sabur, A. Chang, K. Xu, and G. Xue, "IoT system vulnerability analysis and network hardening with shortest attack trace in a weighted attack graph," in *Proc. of ACM/IEEE IoTDI*, 2023.

[12] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia, "An attack graph-based probabilistic security metric," in *Proc. of the 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, 2008.