Scalable Attack Path Finding for Increased Security

Tom Gonda(✉), Rami Puzis, and Bracha Shapira

Department of Software and Information Systems Engineering,
Ben-Gurion University of the Negev, Beer-Sheva, Israel
tomgond@post.bgu.ac.il
http://bgu.ac.il

Abstract. Software vulnerabilities can be leveraged by attackers to gain control of a host. Attackers can then use the controlled hosts as stepping stones for compromising other hosts until they create a path to the critical assets. Consequently, network administrators must examine the protected network as a whole rather than each vulnerable host independently. To this end, various methods were suggested in order to analyze the multitude of attack paths in a given organizational network, for example, to identify the optimal attack paths. The down side of many of those methods is that they do not scale well to medium-large networks with hundreds or thousands of hosts. We suggest using graph reduction techniques in order to simplify the task of searching and eliminating optimal attacker paths. Results on an attack graph extracted from a network of a real organization with more than 300 hosts and 2400 vulnerabilities show that using the proposed graph reductions can improve the search time by a factor of 4 while maintaining the quality of the results.

Keywords: Network security · Attack graphs · Planning · Graph reduction · Attack models

1 Introduction

The software products used in today's corporate networks are vast and diverse [1]. As a result, software vulnerabilities can be introduced to the network which an attacker can later leverage in order to gain control of the organization's hosts. In practice, even organizations that are minded of security can have hosts with many critical vulnerabilities present in their network [2].

One of the security analyst tasks is to decide which vulnerabilities and which hosts to patch against attacks. The cost of patching a host, and the effort involved can some times be extremely high [3]. There is a risk that a patch will break a production system, on top of the maintenance time it takes to patch the system.

This raises the now-common need to prioritize which vulnerabilities in which hosts to patch. An important factor in the decision to patch a host or not is how an attacker can leverage the host as a stepping stone in order to reach

S. Dolev and S. Lodha (Eds.): CSCML 2017, LNCS 10332, pp. 234–249, 2017.
DOI: 10.1007/978-3-319-60080-2_18

critical assets. In order to find the probable path of an attacker, many models have been suggested to represent all attacker's possible paths in a network [4,5].

We chose to use MulVAL (Multi-host, Multi-stage Vulnerability Analysis Language) framework [6] to represent an attacker's possible actions in the network. A brief description of the framework, and the logical attack graphs (Also called LAGs) it produces can be found in Subsect. 3.2.

Using the models that represent the attacker's possible actions, many researchers then applied planning methods to find the optimal attacker's path [7–9]. The downside of many of those methods is that they do not scale well to medium to large networks.

In this paper we aim to reduce the time it takes to find attack paths which an attacker might use, by reducing the size of the attack graph. We intend to do so without effecting the quality of the optimal path. We review the metrics in which we will check that comparison in Sect. 6.

Our contribution is a reduction (described in Sect. 4) that allows finding low-cost attacker paths faster, without compromising the quality of the paths found (experiments in Sect. 7). In results compared to existing approaches on graphs containing more than 200,000 nodes, which represent 309 network hosts with 2398 vulnerabilities the proposed reduction improved the running time in a factor of 4.

2 System Overview

This paper deals with reducing the size of the LAGs, in order to speed the computation time for finding attack paths. Figure 1 shows the overall workflow of our work. First, network scans are being performed to collect data about the network structure and vulnerabilities present in the network as described in Subsect. 3.1. Next, the reductions presented at the related work (Sect. 5) are applied, in order to reduce the input to the graph generation framework (MulVAL). Then, the MulVAL framework is applied to create a logical attack graph. The LAG model is presented in Sect. 3.2. After the LAG was generated, our reduction which is presented in Sect. 4 can be applied in order to reduce the LAG generated in the previous step. At last, the result of the reduced graph is converted to a planning

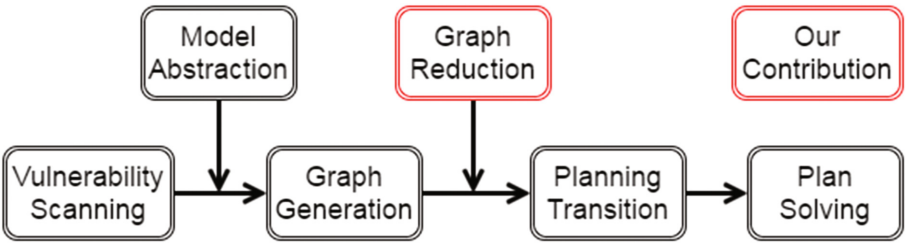

Fig. 1. Work-flow illustration

problem, and solved by a generic planner, as explained in Sect. 3.3. Each experiment described in Sect. 7 will go through all the above steps, although in each experiment only one reduction will be applied, either before or after the graph generation phase.

3 Background

One of the foundations of our work is attack graphs. Attack graphs have been used in multiple variations for over a decade to represent possible attacks on a system [10,11]. Attack graphs usually include information about the preconditions needed to execute actions (exploits, password guessing, network sniffing, etc.) and the possible outcomes of these actions (like agent being installed on target machine, accounts compromised and more). In many cases, attack graphs represent all the attack paths in the target network. Attack path usually represent a series of actions which end with the attacker gaining unauthorized access to an asset. Our main focus is LAGs [12] since they have been the most scalable among the models, and provide open-source implementations.

To produce the attack graph, we had to provide vulnerability scans of the different hosts in the network, and the connections between hosts. We used real-world networks for our work. The way we scanned the networks and produced the topology for the attack graph is outlined in Subsect. 3.1.

We then transformed the attack graph into a planning problem, and used a generic solver to find the optimal attacker path within that attack graph. Scientific background about planning with numeric state variables and the transformation from attack graph to a planning problem can be found in Subsect. 3.3.

3.1 Data Set

In order to create attack graphs as close as possible to the real world, we decided to produce the attack graphs from a large institute with thousands of hosts. For our work we looked at each VLAN in the institute separately. VLAN (Virtual Local Area Network) is a way to unite computers of certain characteristic within an organization. As an example, in corporate network, different departments could be assigned different VLANs so that the sales department and the HR department will have a form of segregation between. In a similar manner, in an academic network, different departments will be assigned different VLANs.

It's a common practice to have a DMZ (demilitarized zone) VLAN, a separate VLAN in which all the services exposed to the Internet will be located. The DMZ VLAN will usually have a restricted access to the rest of the VLANS to minimize the damage an attacker can do in case he compromises a machine in the DMZ.

In order to produce the attack graph we had to find the following information about each VLAN:

1. What are the vulnerabilities in each host in the VLAN?
2. What connections can be made from a VLAN to the rest of the VLANs (some connections can be blocked or enabled through firewall between the VLANs)?

To collect this information we used Nessus vulnerability scanner [13]. We chose Nessus after a comparison with additional vulnerability scanner - OpenVAS [14] since Nessus is more common in attack graph research and has better integration to the MulVAL framework which we used to produce the attack graphs.

We chose 3 different VLANs in the institute which we decided to scan. The scan have been performed in the following manner: First we scanned all the VLANs from the Internet, external to the organization. Then, for two of the VLANs we scanned, we positioned the scanning computer inside the VLAN and scanned the 2 other VLANs, and the VLAN itself from within, An illustration can be seen in Fig. 2.

In order to change the location of the scanning computer between VLANs without the need of physically changing locations we used trunk ports. When using trunk ports, Ethernet frames are tagged with the desired VLAN and then passed to the desired VLAN through ports that are able to handle tagged Ethernet frames. This allowed easier scanning from multiple VLANs without having to physically access the different locations in the organization.

An obvious result we have observed is that scanning a VLAN from different locations produced different results. For example, in some VLANs scanned from the Internet, no hosts were detected. This was probably caused by a firewall filtering connections to this VLAN. In some hosts, we have seen different set of services exposed to different VLANs. For example, when scanning some VLANs from the Internet, only the web service at port 80 was available. When scanning the same host from within the organization we have seen additional services exposed such as web management or network share services.

We used the different scan results achieved when scanning from the different locations to create the topology of the network. For each VLAN, we treated the scan made from within the VLAN as representing the 'true state' of the network. An assumption was made that no device filtered or altered the scan within a VLAN. This is somewhat a possible assumption, since communication within VLAN does not go through any hosts, so the possibility of interference is low. Hence, in the model, the hosts in each VLAN, the services they run and the vulnerability those services have were taken from the scan made from the host inside the VLAN. Before explaining how the connection between hosts were created, a few formal definitions are needed.

Definition 1. *VLAN. A VLAN V is a set of ip addressed such that each ip address is within the VLAN. We denote the different VLANs in the organization as* V_1, V_2, V_3. *For formality, we will define the internet as a VLAN as well, denoted* $V_{internet}$. *In our case* $V_{internet}$ *has only one ip.*

Definition 2. *Connection. A connection c is defined by the tuple:*

$$(ip\_src, ip\_dst, protocol, port)$$

Where ip_src is the source ip of the connection, ip_dst is the destination of the connection. protocol is the network protocol being used (like tcp or udp) and, port is the port used. A connection represent that the source ip can initiate a connecting to the destination ip, in the protocol stated and in the port stated.

Definition 3. *Scan Item. A scan item s is defined by a tuple:*

$$(ip, protocol, port, software, vulnerability)$$

Where ip is an ip of a scanned computer. software is a software installed on the computer and vulnerability is the CVE of the vulnerability found in that software.

Definition 4. *Scan. A scan S_{ij} is a set of scan items. S_{ij} holds:*

$$\forall s \in S_{ij} : p_1(s) \in V_j$$

Where p_n is the n projection of s. A scan represents all the hosts and vulnerabilities found in V_j when scanned from V_i. Since the computer scanning from the internet is irrelevant for us:

$$\forall i \in \{1, 2, 3\} : S_{i,internet} = \emptyset$$

We defined two types of connection: Inner network connection:

$$INNER_i = \{\forall ip \in V_i \quad \forall s \in S_{ii} | (ip, p_1(s), p_2(s), p_3(s))\}$$

Meaning that we model a connection for any two hosts within the VLAN, in all protocols and ports found when scanning the VLAN from within.

Inter network connection:

$$INTER_{ij} = \{\forall ip \in V_i \quad \forall s \in S_{ij} | (ip, p_1(s), p_2(s), p_3(s)\}$$

Meaning that we model a connection between a host in V_i to a host in V_j in some protocol and port only if when scanning V_j from V_i a vulnerability was found in the host at V_j in that protocol and port.

Finally the connections allowed in our model are:

$$K = \{1, 2, 3\}$$

$$\bigcup_{i \in K} INNER_i \cup \bigcup_{i \neq j \in K} INTER_{ij} \cup INTER_{internet,1}$$

We included connection from the internet to only one VLAN in our network because otherwise the graphs produced and solutions found would often be trivial one step solutions.

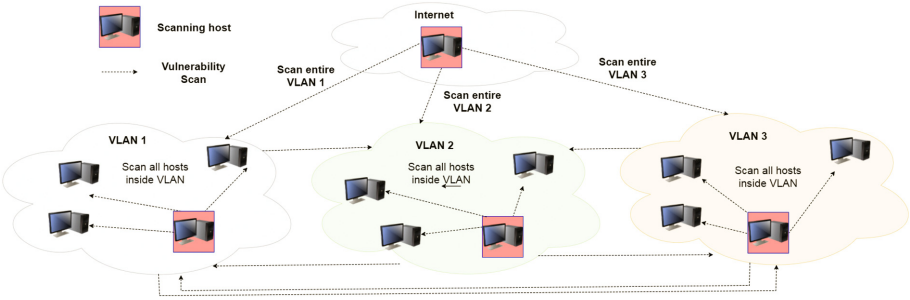

Fig. 2. Scan methodology overview

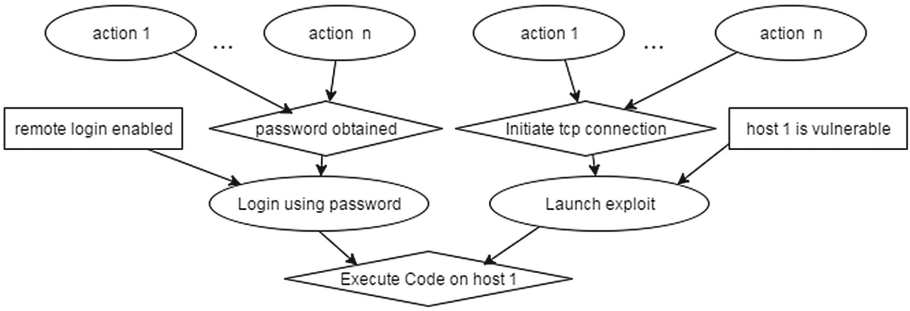

Fig. 3. Example attack graph

3.2 Logical Attack Graph

Logical attack graphs (LAGs) are graphs that represent the possible actions and outcomes of attacker trying to gain a goal asset in a system. The graph contains 3 types of nodes:

Derived fact nodes (can also be referred as privilege nodes) represent a capability an attacker has gained after performing an action (derivation phase). Example of such node can be a node stating that the attacker can execute arbitrary code on a specific machine with certain privileges. These are the diamond shaped nodes seen in Fig. 3.

Derivation nodes (can also be referred as action nodes) usually represent an action the attacker can take in order to gain a new capability in the system. The outcome of performing an action, is an instantiation of a new derived fact. Example of an action node can be seen in Fig. 3 as the oval nodes. One of the possible ways to gain code execution in a host is launching an exploit that allows remote code execution. Another possibility is obtaining a password of a valid user, and logging in with his credentials. A derived fact can be instantiated by **either one** of it's parent nodes, which are action nodes. In order to instantiate an action node (or derivation node) **all** of it's parent nodes need to be instantiated.

Primitive fact nodes are the ground truth nodes of the model, they represent facts about the system. Usually they can represent network connectivity, firewall rules, user accounts on various computer and more. In the example Fig. 3 they are the box shaped nodes.

Definition 5. *Attack Graph. Formally, attack graph is represented as a tuple:*

$$(Np, Ne, Nc, E, L, G)$$

Where N_p, N_e and N_c are three sets of disjoint nodes in the graph, E is a set of directed edges in the graph where

$$E \subseteq (N_e \times N_p) \cup ((N_p \cup N_c) \times (N_e)$$

L is a mapping from a node to its label, and $G \subseteq N_p$ is a set of the attacker goals. N_p, N_e and N_c are the sets of privilege nodes, action nodes and primitive fact nodes, respectively.

The edges in a LAG are directed. There are three types of edges in attack graph: (a, p) an edge from an action node to a predicate node, stating that by applying a an attacker can gain privilege p. (p, a) is an edge from a predicate node to an action node, stating that p is a precondition to action a. (f, a) is an edge from fact node f to an action node a stating that f is a precondition to action a. The labeling function maps a fact node to the fact it represents, and a derivation node (action node) to the rule that is used for the derivation. Formally, the semantics of a LAG is defined as follows: For every action node a, let C be a's child node and P be the set of a's parent nodes, then

$$(\wedge L(P) \Rightarrow L(C))$$

is an instantiation of interaction rule L(a) [12]. In our work we add cost function C to the LAG. $C(a)$ where $a \in N_e$ is the cost the attacker pays to perform an action.

3.3 Planning with Numeric State Variables

Planning is a branch of AI that deals with choosing action sequences in order to achieve a goal. A planning framework is usually given a description of the possible propositions in the world, the possible actions including their preconditions and effects, the initial set of proposition, and the desired propositions in the goal state. It's goal is to find sequence of actions that results in a state that satisfies the goal.

Formally, Numeric planning task is a tuple (V, P, A, I, G) Where P is a set of logical propositions used in the planning task. $V = \{v^1, v^2 ... v^n\}$ is a set of n numeric variables. A state s is a pair $s = (p(s), v(s))$ where $p(s) \subseteq P$ is the set of true proposition for this state, and $v(s) = (v^1(s), v^2(s) ... v^n(s)) \in \mathcal{Q}^n$ is the vector of numeric variables assignments. A is a the set of actions in the problem. An action is a pair $(pre(a), eff(a))$ where pre (precondition) is the precondition needed to be satisfied in order to activate the action. Formally, when planning with numeric states, precondition, con, is also a pair $(p(con), v(con))$ where $p(con) \subseteq P$ is the set of proposition required to be true. $v(con)$ is a set of numeric constraints. In our model, we do not have numeric constraints before activating actions, so we will not go into details about their formal definition. An effect is a triple $eff = (p(eff)^+, p(eff)^-, v(eff))$ where $p(eff)^+ \subseteq P$ is a set of propositions assigned true, as an effect of the action activation, $p(eff)^- \subseteq P$ is a set of propositions assigned false as an effect of the action activation, and $v(eff)$ is a set of effects on the numeric variables in V.

In a numeric planning task, I is a state representing initial state $s = (p(s), v(s))$, and G is a condition representing the goal condition [15]. In our model, attacker's actions comes with a cost (such as risk of detection, or ease of exploitation for vulnerabilities). Our goal is to find a plan with minimal cost for the attacker, assuming an attacker will try to reach his goals with minimal effort or risk.

PDDL is Planning Domain Definition Language, a language build for representing multiple planning problems, specifically it allows modeling numeric tasks.

PDDL also allows specifying optimization criterion, which is an expression the solver will later minimize or maximize. The variable we would like to minimize in our work is the attacker cost for reaching a goal.

Researchers have used planning to represent attacker trying to achieve goals in the network for quite some time [7,9,16]. It seems that the two prominent planners in this domain have been the Metric-FF [15], and SGPlan [17]. In our work we have used Metric-FF planner, since it was able to handle with larger amount of predicates and actions generated when converting attack graphs to a planning problem.

We transformed an attack graph to a planning problem in the following manner: All of the primitive fact nodes have been turned into propositions. Those propositions are initially true in the initial state of the task. All of the derived fact nodes (privilege nodes) where translated into propositions in the model, they are initially false in the model. Each derivation node (action node), became an action $a = (pre(a), eff(a))$ in the planning task. $pre(a) = (p, v)$ where p is a set of the action's precondition, and v is a set numeric constraints. In our model, p contains the proposition of all of the action node's parents in the graph. As an example, in Fig. 3, the action 'Login using password' will have two proposition as preconditions, one that represents the primitive fact 'remote login enabled' and another that represent the derived primitive 'password obtained'.

As stated above, the effect of a is a triple $(p(eff)^+, p(eff)^-, v(eff))$ where $p(eff)^+$ is the predicate representation of the action node's parent. In our example, it will be the predicate of 'Execute Code on host 1'. $p(eff)^- = \emptyset$ since in our current model, the attacker does not lose previously achieved goals by launching new attacks. The numeric effect of an action node is an increase of the numeric variable representing the attacker total effort. Each action node can be assigned with a cost $cost \in \mathcal{N}$. If an action is assigned with a cost, then the numeric effect of the action is: $v(eff) = (total\_effort, +=, cost)$. The goal of the planner is to find a sequence of actions that end in one of the goal predicate true, while minimizing $total\_effort$ variable.

4 PathExpander Algorithm

In this section we will describe our proposed reduction algorithm. In the core of the algorithm we find the shortest path between a source node and a target node, and expand the graph using this shortest path. For this, we assume that our graph has a source node, and that the graph has a single target node. We argue that these are valid assumptions in our model. For source node, all the attack graphs have a fact node representing the attacker initial location. This can be used as the source node for our algorithm. For target node, we choose the goal node in the LAG. In case there are multiple goal nodes in the attack graph, we can easily create a single goal by creating virtual actions applicable only from goal nodes, which lead to a single new goal node. This transition was described in depth in [18].

After we have the shortest path between a goal node and a source node in the LAG, we verify that all of the action node's preconditions are met in that path.

Algorithm 1. PathExpander algorithm

```
1  function PathExpander(G, s, t)
   Input  : LAG G, source s, target t
   Output: Reduced LAG G'
2  forall v ∈ G do
3  |  Color(v) = White;
4  end
5  Q ← WeightedShortestPath(G, s, t);
6  while Q ≠ ∅ do
7  |  v ← Q.pop();
8  |  if Type(v) = fact then
9  |  |  Color(v) ← Black;
10 |  else if Type(v) = action then
11 |  |  if Color(v) = White then
12 |  |  |  Color(v) ← Grey;
13 |  |  |  Q.push(v);
14 |  |  |  if ∃u ∈ v.parents s.t Color(u)=Grey then
15 |  |  |  |  Continue;                          /* Loop Detected */
16 |  |  |  else
17 |  |  |  |  forall u ∈ v.parents do
18 |  |  |  |  |  Q.push(u);
19 |  |  |  |  end
20 |  |  |  end
21 |  |  else if Color(v) = Grey ∧ ∀u ∈ v.parents : Color(u) = Black then
22 |  |  |  Color(v) ← Black
23 |  else if Type(v) = privilege then
24 |  |  if ∃u ∈v.parents s.t Color(u) = Black then
25 |  |  |  Color(v) ← Black;
26 |  |  |  continue;
27 |  |  if Color(v) = White then
28 |  |  |  Color(v) ← Grey;
29 |  |  Q.push(v);
30 |  |  U = FilterGreyNodes(v.parents);
31 |  |  Q.push(GetMinimumNode(U));
32 end
33 return Subgraph of G induced by black nodes
```

A precondition to an action node can be either derived predicate (privilege node) or a primitive fact node. In case it's a primitive fact node, we simply add that node to the reduced graph. In case it's privilege node, we have to decide which action will satisfy that node. We choose to expand the action with minimal cost that can satisfy the privilege node. Careful care should be taken in order to handle possible cycles in the graph. We have solved this complexity by incorporating a mechanism similar to the DFS search algorithm.

The algorithm is specified in Algorithm 1. In line 5, we assign a stack data structure, Q, with the shortest path in the attack graph G, where the source node

is at the top of the stack, and the goal node is at the bottom. If we encounter a leaf node (fact node) we can't expand that node further, and mark it as a solved node in line 9. If we encounter an action node for the first time, we add it to the stack, to revisit and make sure all his parent nodes were also satisfied. If one of the action node's parents was visited already (grey) this means we encountered a cycle and should not mark this action node as resolved. If all of the action node's children were resolved (black), we can resolve this action node (line 22). For privilege node, we first check if one of it's parents (action nodes) is satisfied (line 24). If so, then the action node also satisfied the current privilege node (line 25). If the privilege node is not already satisfied by an action node, we expand the privilege node's cheapest parent (which is an action node) that is not already expanded (lines 30 and 31).

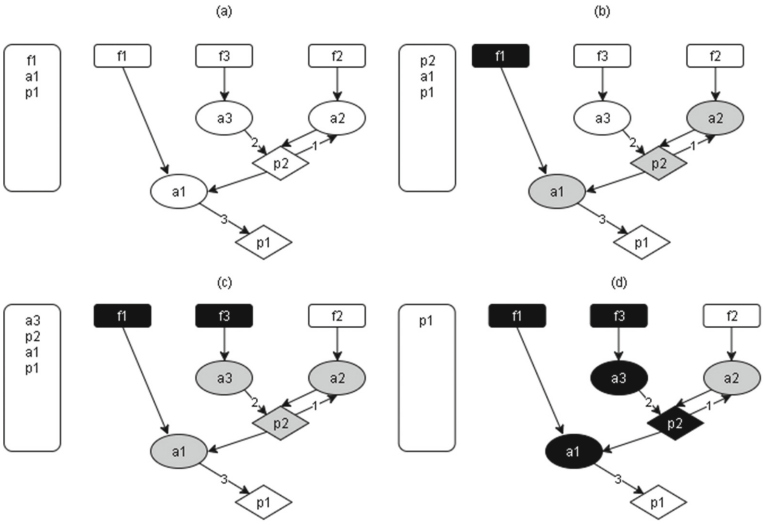

Fig. 4. Example PathExpander execution. Source node is f1 and destination node is p1

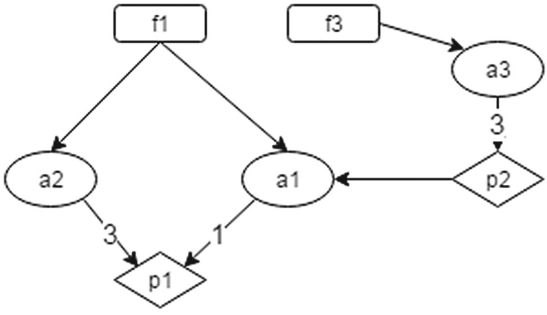

Fig. 5. Example LAG for which PathExpander algorithm returns non-optimal sub-graph

In Fig. 4, the goal node is p1, and the source node is f1. First the shortest path is found. In this case the shortest path includes f1 → a1 → p1 - Fig. 4a. We pop f1, it's a leaf node, so we mark it black, and continue to the next node. We pop a1, it's not yet visited so we mark it gray, and push it back to the stack to revisit. a1 has one parent node, p2 so we push p2 to the stack. p2 is poped, it is a privilege node that has no black children, so we continue. It's color is white, so we mark it gray. We push p2 back since we'll have to revisit and make sure p2 was satisfied. p2 has two parents: a2 and a3. The edge (a2, p2) is cheaper than (a3, p2) so we push a2 to the stack. After we pop a2, we notice it has a gray parent node (p2) meaning a cycle. So we skip a2, after marking him gray (Fig. 4b). Since we pushed p2 before pushing a2, we will pop p2 again. This time, p2 has only one white parent node - a3. So we push a3. This time, all of a 3's children can be satisfied (f3, Fig. 4c). So we satisfy a3, and then when we revisit p2 again we notice it has a black (satisfied) parent, and satisfy p2 as well. We now pop a1, this time, all of his parent nodes are black (f1 and p2), so we mark it black too (Fig. 4d). We pop p1, and it has a black child node: a1, hence we will mark p1 as satisfied as well. We will return the sub-graph induced from the nodes f1, a1, p2, a3, f3, p1 and the edges between them in the original graph.

This sub-graph will not always be optimal. For instance, in Fig. 5, for source node f1 and destination node p1, the resulting subgraph will contain the nodes: f1, a1, p1, p2, a3, f3 with cost 4. While the cost for the sub-graph from the nodes: f1, a2, p1 will be 3. Experiment show that the path expander algorithm often returns a sub-graph that contains the optimal solution.

5 Related Work

Since LAGs were used to illustrate all possible paths an attacker can take in order to compromise the network, it became apparent that these graphs are often complex and difficult to comprehend fully. A human user may find it problematic to reach appropriate configuration decisions looking at these attack graphs.

For this reason, many researchers have set their goal to reduce the size of a LAG, with minimal impact to the conclusion that can be drawn from the reduced attack graph [19–21]. Zhang et al.'s work is the only reduction that could directly be used on LAGs and that the reduction outcome can be transformed into a planning problem. Similar methods have been proposed for Multiple Prerequisites (MP) graphs [11].

5.1 Effective Network Vulnerability Assessment Through Model Abstraction

In their work, Zhang et al. [21] suggest that the graph reduction will take place before the attack graph is generated. The steps to achieve this reduction are:

1. Reachability-based grouping. Hosts with the same network reachability (both to and from) are grouped together.

2. Vulnerability grouping. Vulnerabilities on each host are grouped based on their similarities.
3. Configuration-based breakdown. Hosts within each reachability group are further divided based on their configuration information, specifically the types of vulnerabilities they possess.

Following those steps results in an reduced input to attack graph generators - namely MulVAL, which results in a reduced and easier to understand attack graph. In an experiment described in the article, an attack graph with initially 217 nodes and 281 edges was reduced to 47 nodes and 55 edges. In our experiments we also applied those algorithms with some success. In their work, the authors also examined the effect such reductions have on the quantitative security metrics of the attack graph which represent the likelihood an asset will be compromised [22]. It was shown that using this reductions yields different security metrics for different hosts in the network, compared to the original model. The authors claimed that the new security metrics represent the real world better, since many of the vulnerabilities are dependent of each other. We implemented some of the reductions described here, and tested their effectiveness (This will be described in the results section).

6 Evaluation

Our goal is to evaluate how different reductions affect two main parameters. The first parameter is the time it takes finding minimal attack path.

$$TotalTime = Gen + Reduce + Solve$$

Where *Gen* is the time it takes MulVAL framework to generate an attack graph, *Reduce* is the running time of the reduction algorithm and *Solve* is the time it takes the solver to find a solution. The second parameter we took into account is the cost of the minimal plan found by the solver using the different reductions.

$$TotalCost = \sum_{a \in P} Cost(a)$$

Where P is the plan action sequence found by the solver, and $Cost(a)$ is the cost of an action in the sequence according to our attack graph. Initially, the costs in our experiments were taken from exploitability metric in CVSS (Common Vulnerability Scoring System) [23] to represent the easiest exploitable path in the graph. Meaning the path found is the easiest exploitable path for an attacker.

After some experiments we have noticed that the $TotalCost$ of all the paths found have the same cost which is the number of steps an attacker takes. By investigating the results we have concluded that the vulnerabilities costs using the exploitability metric lack variance. To illustrate: more than 70% out of 2500 vulnerabilities were of cost 1 and 2 (the easiest exploits). In order to produce more varied data, we have randomly assigned the cost for vulnerabilities in

our experiments, drawn from a uniform distribution. To test the reductions on datasets with different sizes, we created 4 additional datasets from the original dataset, in which only vulnerabilities above certain CVSS impact metric (representing the damage an attacker can cause by applying a vulnerability) were included. This created 5 different datasets with varying number of hosts and vulnerabilities, and varying cost for vulnerabilities.

7 Results

Figure 6 shows the running time in seconds it took to find the attacker path for each reduction on networks in different sizes. The Y axis, in log scale, shows the overall time it took find an attack path. This includes the time it took to generate the attack graph, the time it took to reduce the attack graph and the time it took the planner to solve the planning problem. "Without" is the baseline, meaning that we do not change the original LAG in any form. "Grouping" Refers to the 1st reduction presented in Subsect. 5.1 in which hosts with similar reachability configuration are grouped. "Aggregate" refers to the 2nd reduction in Subsect. 5.1 in which similar vulnerabilities in a software installed on a host are aggregated together. "Aggregate and Group" means applying the two previous reductions together. "PathExpander" is our algorithm described in Sect. 4. The X axis, in log scale shows the number of nodes in the attack graph. The largest graph which included all the hosts and vulnerabilities had 220,700 nodes and represented 309 hosts containing 2398 software vulnerabilities. The results show that as the size of the network gets bigger, PathExpander algorithms finds an attacker path about 4 times faster than the second best reduction (Aggregate and Group). The trend-line for the PathExpander is $y = 0.3892x + 3.9922$ with $R^2 = 0.9939$ while the trend-line for the Aggregate and Group reduction is $y = 1.5598x - 8.59$ with $R^2 = 0.9622$.

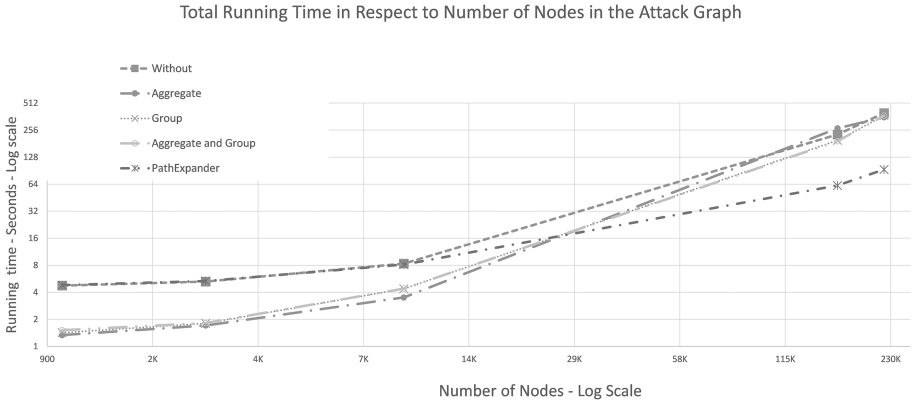

Fig. 6. Total run time in respect to the size of the network and reduction used

Figure 7 shows the cost of the plan found using each reduction in respect to the size of the network. As we can see, using different reductions we found plans with different costs. This is possible due to the fact the planner we have used, Metric-FF is not an optimal planner, and does not guarantee to return the optimal plan. Another important fact we notice is that sometimes by reducing the size of the graph, we find better paths than those found in the non-reduced graph. We have manually checked the planning input files in those cases and made sure that the low-cost plan found on the reduced graph were present in the non-reduced graph, and indeed they were present.

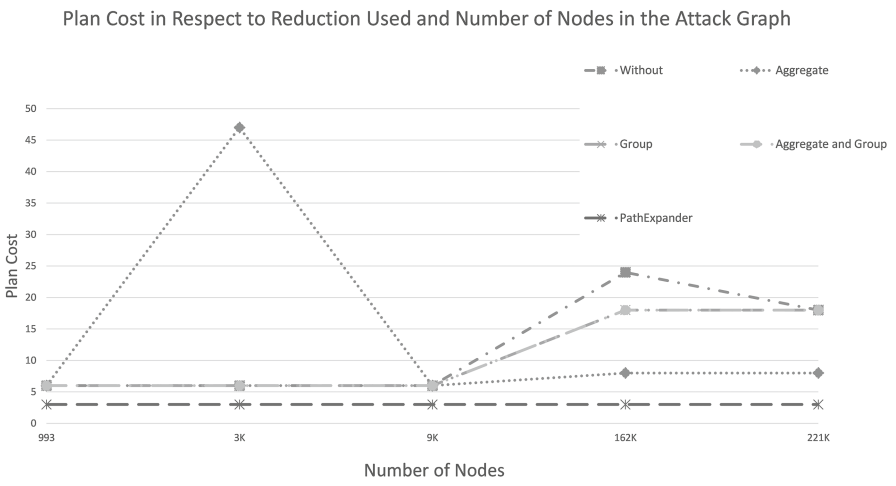

Fig. 7. Total cost in respect to the size of the network and reduction used

8 Conclusion and Discussion

By looking at the results, we can see two interesting trends which are desired for our method. First, in terms of running time, we observe that as the network gets bigger, PathExpander finds solutions much faster than the compared methods. In the largest network which contained 220,700 nodes and represented 309 network hosts with 2398 vulnerabilities, our method found a result more than 3 times faster (93.34 s compared to 356.91 s) than the second best reduction used (Aggregate and Group).

In terms of the quality of the results, our methods consistently found the best attacker path compared to the other methods. We suspect that this is due to the fact that the planner we have used, Metric-FF is not an optimal planner.

Those two results show that using the PathExpander algorithm in order to reduce an attack graph before searching for solution using general planner can both improve the overall running time it takes to find an attacker path, and the

quality of the paths found. This can allow security administrators derive better conclusions in regards to which vulnerabilities in which hosts to patch first in order to keep the network secure.

Although in our experiments, drawn from real-life scenarios, the cost of the paths found were always optimal, in the general case this might not always be true. In the future we aim to analyze the conditions in which the results are guaranteed to be optimal. More-over we intend to examine how similar reduction methods can be applied to more complex models that include both costs for actions and probabilities of success.

Another possibility for future work is to relax the assumptions about the attacker. Mainly the fact that this work assumes that the attacker knows the networks structure and is aware of the target assets. Works such as [24,25] have started examining this topic, and the question stands how PathExpander can be applied in those models.

References

1. Morrow, B.: Byod security challenges: control and protect your most sensitive data. Netw. Secur. **2012**(12), 5–8 (2012)
2. Zhang, S., Zhang, X., Ou, X.: After we knew it: empirical study and modeling of cost-effectiveness of exploiting prevalent known vulnerabilities across IaaS cloud. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, pp. 317–328. ACM (2014)
3. Shostack, A.: Quantifying patch management. Secure Bus. Q. **3**(2), 1–4 (2003)
4. Ammann, P., Wijesekera, D., Kaushik, S.: Scalable, graph-based network vulnerability analysis. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, pp. 217–224. ACM (2002)
5. Sheyner, O.M.: Scenario graphs and attack graphs. Ph.D. thesis, US Air Force Research Laboratory (2004)
6. Ou, X., Govindavajhala, S., Appel, A.W.: MulVAL: a logic-based network security analyzer. In: USENIX Security (2005)
7. Roberts, M., Howe, A., Ray, I., Urbanska, M., Byrne, Z.S., Weidert, J.M.: Personalized vulnerability analysis through automated planning. In: Working Notes of IJCAI 2011, Workshop Security and Artificial Intelligence (SecArt 2011), vol. 4 (2011)
8. Sarraute, C.: New algorithms for attack planning. In: FRHACK Conference, Besançon, France (2009)
9. Ghosh, N., Ghosh, S.: An intelligent technique for generating minimal attack graph. In: First Workshop on Intelligent Security on Security and Artificial Intelligence (SecArt 2009). Citeseer (2009)
10. Poolsappasit, N., Dewri, R., Ray, I.: Dynamic security risk management using Bayesian attack graphs. IEEE Trans. Dependable Secure Comput. **9**(1), 61–74 (2012)
11. Ingols, K., Lippmann, R., Piwowarski, K.: Practical attack graph generation for network defense. In: 22nd Annual Conference on Computer Security Applications Conference, ACSAC 2006, pp. 121–130. IEEE (2006)
12. Ou, X., Boyer, W.F., McQueen, M.A.: A scalable approach to attack graph generation. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 336–345. ACM (2006)

13. Beale, J., Deraison, R., Meer, H., Temmingh, R., Walt, C.V.D.: Nessus Network Auditing. Syngress Publishing, Rockland (2004)
14. OpenVAS Developers: The Open Vulnerability Assessment System (OpenVAS) (2012)
15. Hoffmann, J.: The Metric-FF planning system: translating "ignoring delete lists" to numeric state variables. J. Artif. Intell. Res. **20**, 291–341 (2003)
16. Obes, J.L., Sarraute, C., Richarte, G.: Attack planning in the real world. arXiv preprint arXiv:1306.4044 (2013)
17. Chen, Y., Wah, B.W., Hsu, C.W.: Temporal planning using subgoal partitioning and resolution in SGPlan. J. Artif. Intell. Res. **26**, 323–369 (2006)
18. Albanese, M., Jajodia, S., Noel, S.: Time-efficient and cost-effective network hardening using attack graphs. In: 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 1–12. IEEE (2012)
19. Noel, S., Jajodia, S.: Managing attack graph complexity through visual hierarchical aggregation. In: Proceedings of the 2004 ACM Workshop on Visualization and Data Mining for Computer Security, pp. 109–118. ACM (2004)
20. Homer, J., Varikuti, A., Ou, X., McQueen, M.A.: Improving attack graph visualization through data reduction and attack grouping. In: Goodall, J.R., Conti, G., Ma, K.-L. (eds.) VizSec 2008. LNCS, vol. 5210, pp. 68–79. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85933-8_7
21. Zhang, S., Ou, X., Homer, J.: Effective network vulnerability assessment through model abstraction. In: Holz, T., Bos, H. (eds.) DIMVA 2011. LNCS, vol. 6739, pp. 17–34. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22424-9_2
22. Homer, J., Ou, X., Schmidt, D.: A sound and practical approach to quantifying security risk in enterprise networks. Kansas State University Technical Report, pp. 1–15 (2009)
23. CVSS: A complete guide to the common vulnerability scoring system (2007)
24. Shmaryahu, D.: Constructing plan trees for simulated penetration testing. In: The 26th International Conference on Automated Planning and Scheduling, vol. 121 (2016)
25. Hoffmann, J.: Simulated penetration testing: from "Dijkstra" to "turing test++". In: ICAPS, pp. 364–372 (2015)