Ranking Vulnerability Fixes Using Planning Graph Analysis

Tom Gonda, Guy Shani, Rami Puzis, Bracha Shapira
SISE Department
Ben Gurion University, Israel
{tomgond,shanigu,puzis,bshapira}@bgu.ac.il

Abstract

During the past years logical attack graphs were used to find the most critical vulnerabilities and devise efficient hardening strategies for organizational networks. Most techniques for ranking vulnerabilities either do not scale well, e.g. brute-force attack plan enumeration, or are not well suited for the analysis of logical attack graphs, e.g. centrality measures.

In this paper we suggest an analysis of the planning graph (from classical planning) derived from the logical attack graph to improve the accuracy of centrality-based vulnerability ranking metrics. The planning graph also allows efficient enumeration of the set of possible attack plans that use a given vulnerability on a specific machine. We suggest a set of centrality based heuristics for reducing the number of attack plans and compare with previously suggested vulnerability ranking metrics. Results show that metrics computed over the planning graph are superior to metrics computed over the logical attack graph or the network connectivity graph.

1 Introduction

Large organizations use a vast and diverse set of software [Morrow, 2012]. As such, ensuring that all installed software are completely safe is an impossible task. The computer networks of large organizations can hence be penetrated by exploiting vulnerabilities in the installed software, operating system, or their combinations. Indeed, research has shown that even organizations whose core business is in developing security software have many vulnerabilities in their networks[Zhang *et al.*, 2014].

These vulnerabilities can often be fixed. For example, when a vulnerability in a given program is identified, the software company maintaining the software often issues a patch fixing the vulnerability. Alternatively, if a given software is found to be too vulnerable, the security administrator can choose to move from Windows XP to Windows 10, or to move from Windows to Linux or vice versa. Of course, replacing the software often involves a significant cost [Shostack, 2003]. When moving from Windows to Linux one has to install software versions appropriate to Linux instead of the Windows versions. Thus, the system administrator must prioritize the fixes such that the more important vulnerabilities will be fixed first [Cukier and Panjwani, 2009].

Most research focuses on analyzing possible attacks on the network in order to rank the vulnerability fixes. A common data structure for conducting such analysis is the logical attack graph (LAG), whose nodes represent assets or vulnerability exploits, and edges represent which assets are needed before an exploit can be used, or which assets an exploit produces [Ou *et al.*, 2005]. One can analyze the attack graph [Albanese *et al.*, 2012] to gain better understanding of which vulnerabilities can be used to gain a specific sensitive information from a given starting point (e.g. when controlling only machines outside the organization).

Hoffmann et al [Hoffmann, 2015] suggested a different approach for identifying vulnerabilities, by computing directly attack plans for penetration testing (pentesting). An attack plan is a sequence of actions (e.g. exploits) that allow the attacker to achieve its goals, such as access to specific sensitive information. The system administrator can use these attack plans to decide which vulnerabilities to patch. We suggest taking this approach to the extreme, computing all possible attack plans together. Then, the set of all attack plans can be used for, e.g., identifying vulnerabilities that appear in more plans, ranking their fixes higher.

We explain how the (relaxed) planning graph — a data structure often used in the classical planning community, mainly to compute forward search heuristics — can be used to compute the set of all possible plans in our application. We provide an algorithm for enumerating all such plans using a backward scan of the planning graph.

Previous research has suggested various node centrality measures, such as betweenness [Hong and Kim, 2013], and pagerank [Sawilla and Ou, 2008], for ranking the vulnerabilities to be fixed. We demonstrate here that over a range of benchmarks including a scan of a large organization network, metrics computed over the planning graph provide a much better ranking compared to metrics computed over the logical attack graph. We focus here on the task of increasing the cost of the minimal attack. We rank different vulnerabilities by the various metrics, and show that ranking using metric over the planning graph increases the minimal attack cost compared

to rankings based on the logical attack graph. Moreover, for this task, pagerank has shown the best results.

2 Background

We now briefly review relevant background, starting with attack graphs and their use in ranking vulnerability fixes, and then discussing the planning graph data structure, and how it is used for classical planning in general, and for pentesting in particular.

2.1 Logical Attack Graphs

Logical attack graphs (LAGs) are graphs that represent the possible actions and outcomes of actions applied by an attacker trying to gain a goal asset in a system. An example of an attack graph can be seen in Figure 1.

We now describe the attack graph structure. The graph contains 3 types of nodes:

1. **Primitive fact nodes** represent facts about the system. For example, they can represent network connectivity, firewall rules, user accounts on various computer and more. In the example graph (Figure 1) primitive fact nodes are represented by rectangular nodes.
2. **Derivation nodes** (or action nodes) 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. Action nodes are represented in figure 1 by ovals.
3. **Derived fact nodes** (or privilege nodes) represent a capability an attacker gains after performing an action (derivation phase). For example, a node stating that the attacker can execute arbitrary code on a specific machine with certain privileges. Derived fact nodes are represented by diamonds in Figure 1.

Edges in the LAG from a fact node to an action node represent the dependency of the action on the facts, and edges from an action to fact represent the derivation of that fact following the action.

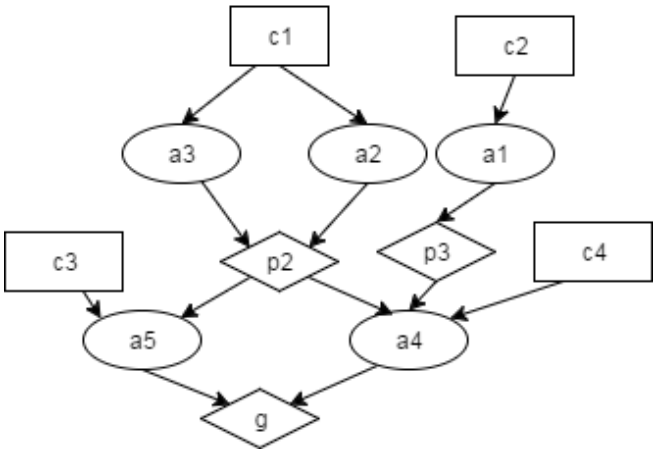


Figure 1: Example of an attack graph

Definition 2.1. *Logical attack graph. Formally, a logical attack graph is a tuple:*

$$G = (N_p, N_e, N_c, E, L, g)$$

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 $g \in N_p$ is the attacker's goal (multiple goals can be transformed into a single goal using an action with preconditions as the multiple goals). N_p, N_e and N_c are the sets of privilege nodes, action nodes and primitive fact nodes, respectively.

The edges in an LAG are directed. There are two types of edges in attack graph: (a, p) an edge from an action node to a derived fact node, stating that by applying a an attacker can gain privilege p. (p, a) is an edge from a fact (either primitive or derived) node to an action node, stating that p is a precondition to action a. For example, in order to apply exploit e on machine m_2 from machine m_1, there must be a connection from m_1 to m_2 (represented by a primitive fact node p), and the user must have already gained access to code execution on m_1 (represented by a derived fact node d). Hence, there will be edges from p to e and from d to e. In addition, if using exploits e results in obtaining code execution privileges on m_2, represented by a derived fact node c, then there will be an edge from e to c.

The labeling function maps a fact node to the fact it represents, and an action node to a rule that defines the derivation of new facts. Formally, 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). [Ou *et al.*, 2006] LAGs are a special case of And/Or Graphs [De Mello and Sanderson, 1990] where each action can instantiate only one fact (or derived fact). We will use this notation from [Gefen and Brafman, 2012]

- $pre(a) = \{v \in N_p \cup N_c : (v, a) \in E\}$
- $add(a) = \{v \in N_p : (a, v) \in E\}$
- $ach(v) = \{a \in N_e : v \in add(a)\}$

Where $pre(a)$ is the set of facts which are preconditions to the action a. $add(a)$ is the set of facts gained by applying the action a (in LAGs this set contains only one node). $ach(v)$ is set of actions which can achieve derived fact node v.

An attack plan G' is a subgraph of G. The attack plan must hold the following:

- $g \in G'$
- $\forall a \in G'_{N_e} : pre_G(a) \subseteq G'$
- $\forall v \in G'_{N_p} : \exists a \in ach_G(v) s.t a \in G' \wedge |ach_{G'}(v)| = 1$

Meaning, an attack plan is a sub-graph of G' that contains the goal node of graph G. Each action a in G' is fulfilled by all of the preconditions of a in G. Each fact is achieved by exactly one action. Attack plan represents a scenario in which an attacker infiltrates the organization and achieves his goals.

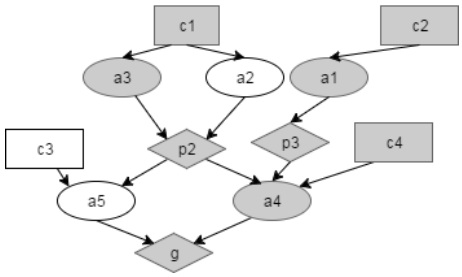


Figure 2: Example attack plan of Graph G in Figure.1

2.2 Graph Centrality Measures

Graph centrality is a sub-field in graph theory research. Centrality measures try to capture how important a node is within a graph. This is useful in many domains, such as social network (finding prominent members in social networks) and more.

Previous research on ranking vulnerability fixes has used some centrality measures in order to identify which vulnerabilities should be fixed [Hong and Kim, 2013] [Sawilla and Ou, 2008] following is a list of such measures that were previously used to that effect.

One of the basic centrality measure in graphs is the degree centrality. mainly because it is easy to compute. Although it is easy to compute ($O(1)$),degree centrality often poorly represents the true importance of a node in a graph.

$$C_D(v) = degree(v)$$

Betweenness centrality captures a more delicate aspect of the importance of the node in a given graph. This measure represents for each node, the number of shortest paths between any two nodes that passes through that node. Formally, betweenness is:

$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Where σ_{st} is the number of shortest paths between nodes s and t, and $\sigma_{st}(v)$ is the number of shortest paths between nodes s and t that pass through node v. It was used in few researches aiming to find important nodes in attack graphs [Hong and Kim, 2013]. The C_B values of the nodes can be computed using [Brandes, 2001] in $O(|V||E|)$.

We have also used a variation of betweenness centrality in which only paths starting from a subset of the nodes in the graph, or ending in a subset of nodes in the graph were counted.

Another commonly used graph centrality measure is the Closeness Centrality. This centrality measure captures how close a certain node is to the rest of the nodes in the graph. In this centrality method, nodes on the fringe of the graphs should score lower than nodes in the center of the graph. Formally closeness centrality is defined by:

$$C_C(v) = \frac{1}{\sum_u d(u,v)}$$

Where $d(u,v)$ is the shortest distance between u and v. The running time for finding C_C is $O(nm + n^2 log(n))$ where n is the number of nodes in the graph, and m is the number of edges in the graph [Wang, 2006].

Researches also used Google's PageRank to rank important nodes in a graph. Initially used to rank the importance of web pages in the Internet, PageRank's essence is measuring how likely for a web-surfer to be at page i [Page *et al.*, 1999]. $0 < d < 1$ is a damping factor, representing how likely a web surfer will get bored and move to another web page which is not directly linked to the current node.

The metric is given by:

$$C_{PR} = \frac{1-d}{N} + d \sum_{j \in In(j)} \frac{\pi_j}{|Out(j)|}$$

Where N is the number of nodes in the graph, $Out(j)$ are the outgoing neighbors of j, $In(j)$ are the ingoing neighbors of j, and π_j is the probability that the web-surfer will be at nodes j. Fast, distributed algorithms for approximation of the PR values exists [Sarma *et al.*, 2015]. This algorithm runs in $O(\frac{logn}{\epsilon})$ rounds. where n is the number of nodes in the network and ϵ is the damping factor.

2.3 Planning Graphs

Planning graphs [Blum and Furst, 1997] are a data structure from the automated classical planning community. A planning graph is a directed, layered graph with two types of nodes and two kinds of edges. The layers change between fact layers, containing only fact nodes, and action layers containing action nodes.

In general planning problems, already obtained facts can be removed by other actions. Planning graphs hence include additional information, such as which facts cannot be achieved at the same time (mutexes). However, in pentesting, once a fact is obtained, it is never lost. We can hence focus on the relaxed planning graph, where obtained facts cannot be lost, which is much simpler to represent and reason about.

The first layer of the relaxed planning graph is a fact layer, and contains one node for each condition node $c \in G_c$. The next layer is an action layer, containing all actions that can be executed using the facts at the previous layer. That is, all actions whose preconditions appear in the previous layer. The third layer contains all the effects of the actions at the second layer.

In addition, we add for each fact p a special *no-op* action, that takes p as precondition, and generates p as output. Hence, each fact layer is a superset of the preceding fact layer. Once no new facts have been obtained in a fact layer, we can stop the expansion of the planning graph.

Edges in a planning graph represent relations between actions and facts. The action nodes in action-layer i are connected by "precondition-edges" to their preconditions in fact layer i. The action nodes are also connected to their add-effects facts in layer $i+1$ by "add-edges".

Action nodes may exist at layer i only if all of their preconditions exist at fact layer i. A fact may exist at fact-layer $i+1$ if it is an effect of some action in action layer i. Thus, the planning graph avoids cycles by allowing repeated fact and action nodes at different layers.

Facts often appear in multiple layers in the planning graph — once a fact has appeared at layer i, it will appear in all fact layers $j > i$. We denote each fact by its layer, that is, for fact p at layer i, we write p_i.

Figure 3 shows a planning graph for the graph G presented in Figure 1. We omit some of the edges between the facts and "no-op" actions for ease of presentation.

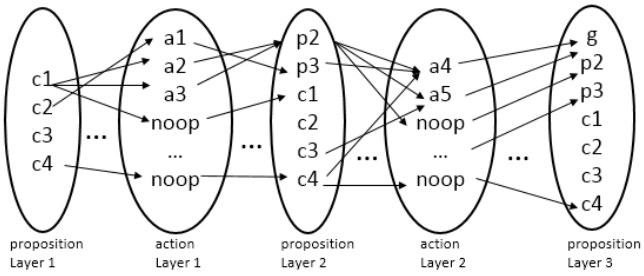


Figure 3: Planning graph of Graph G from Figure.1

To conclude, planning graphs in delete-free domains capture the same information as an attack graph, in a slightly different format. Below, we suggest analyzing the planning graph, replacing the standard centrality measures with an analysis of all attack plans.

3 Enumerating All Attack Plans

We now explain how one can use the planning graph in order to enumerate all possible attack plans. Once we have enumerated all possible attacks, we can, e.g., identify fact or action nodes that participate in many such plans, which may be good candidates for an early fix.

We analyze the planning graph, rather than the LAG, because LAGs contain cycles, which are avoided in the planning graph by using repeated nodes. We use a BFS-style algorithm, moving backward from the goal node g_n at the last fact layer n.

We maintain a set of plans. For each plan there is a set of unsatisfied facts, initialized with the goal. To expand a plan backwards from layer i, for each unsatisfied fact p_{i+2}, we identify an action a (possibly a no-op) that has p in its effects. We remove p_i from the list of unsatisfied facts, and for each fact q in the preconditions of a we add q_i to the set of unsatisfied facts. If a provides an additional unsatisfied fact r_i, it is also removed from the list of unsatisfied facts. That is, we will not search for another action a' to satisfy r_i.

There can be many potential actions that satisfy a needed fact p, each corresponding to a different plan. Thus, for each action a that satisfy p we create a copy of the plan and add a to the copy. Thus, the expanded plan is split into multiple identical plans, differing on the last added action only.

More precisely, let P_i be the set of unsatisfied facts of the expanded plan at layer i, and $\mathcal{A}^P_{i-1} = \{a : \exists p \in P_i, p \in effects(a)\}$ be the set of actions at layer $i-1$ that satisfy at least one fact in P_i. We create a copy of the plan for each minimal subset $A^P_{i-1} \subseteq \mathcal{A}^P_{i-1}$ such that $P_i = \bigcup_{a \in A^P_{i-1}} effects(a)$, and add A^P_{i-1} to the copy.

Algorithm 1: Enumerating All Attack Plans

```
EnumeratePlans(PG, t) :
    Input: Planning graph PG, target node t
    Output: Set of all the attack plans in the graph
    P ← {t}; // Solution plans
    cur_layer ← lastLayer(PG);
    while cur_layer ≠ 0 do
        if cur_laye.type = action then
            for p ∈ P do
                for a ∈ p.cur_layer_nodes do
                    p ← p ∪ a.predecessors;
                end
            end
        end
        else if cur_laye.type = fact then
            for p ∈ P do
                P ← P \ p;
                FA ← ∅; // open fact nodes achievers
                for f ∈ s.cur_layer_nodes do
                    FA ← FA ∪ {f.predecessors};
                end
                AA ← CartesianProduct(FA);
                for ActionSet ∈ AA do
                    P ← P ∪ (p ∪ ActionSet);
                end
            end
        cur_layer ← cur_layer − 1;
    end
    Return(P);
end
```

Once we have reached the initial layer we have enumerated all possible plans. Let Π be the set of all such plans. Π may contain some redundancies, due to the use of no-ops. More specifically, given $P_i = \{p_i, q_i\}$, and two action a_p, a_q that produce p, q, respectively, we may have 4 different alternatives — $\langle a_p, a_q \rangle$, $\langle a_p, noop_q \rangle$, $\langle noop_p, a_q \rangle$, and $\langle noop_p, noop_q \rangle$ for expanding the plan backwards. Then, at layer $i-2$, we can choose a_p where $noop_p$ was selected and a_q where $noop_q$ was selected. Ignoring the no-ops, which are not real actions to be executed, we obtain 4 identical plans. To remove such duplicates, once we have obtained the set of all plans, we remove no-ops from all plans, and then remove duplicate plans, ignoring the action order within a plan.

Using the above planning graph construction and plan enumeration method only yields plans with bound number of actions (which is the number of action layers in the planning graph). In order to allow plans in various lengths, additional edges should be added to the planning graph between the final fact layer and the final actions layer. At this point we have chosen to use only the plans with the shortest length, and not allow plans with larger amount of actions that needed.

This process is obviously np hard, but in the real world graph that we have obtained, it runs sufficiently fast to be useful. Creating the planning graph and enumerating the plans took less than a second on both graphs Table. 1. The ex-

Dataset	Nodes	Plans	Time(Seconds)
localPlus	394	48	0.04
ScannedNetwork	1013	2012	0.52

Table 1: Enumeration running time in respect to LAG size

periments where performed on a Virtual Machine using one Intel Xenon E5-2620 v2 @ 2.10 GHz processor, with 8 GB of RAM. In the future, we will explore sampling techniques, originating from research in AND-OR graphs, to provide a rapid estimation of the needed statistics.

Using the set of all plans we can compute useful measurements. For example, we can count for each action a the number of plans in which a participates:

$$C_{\Pi}(a) = |\{\pi : \pi \in \Pi, a \in \pi\}|$$

We can also compute this over a subset of plans, such as only over the set of plans of length at most k, thus identifying actions that appear in shorter, more efficient, plans.

4 Network Data Acquisition

To test our approach we created realistic models using data obtained from scanning the network of a large organization, containing several subnets. Using the machine configurations and existing exploits discovered using the scan, we can create real world models that allow us to provide an empirical evaluation of our approach. We now provide some explanations of the model and the network, unfortunately omitting many details due to confidentiality restrictions.

To collect the needed information for our models, we began by running a scan of the various subnets using the Nessus scanner [Beale *et al.*, 2004]. Nessus starts its scan from a given computer, and identifies all reachable hosts from that computer, including desktops, gateways, switches, and more.

As Nessus does not actually launch attacks to control a host, a Nessus scan identifies only hosts that are directly logically reachable from the source machine where the scan is running, possibly through several switches and gateways. We hence executed several such scans, each from a different subnet within the organization, as well as one scan from outside the organization network.

The resulting scans contain the set of machines that are visible from each source machine. The machines inside a subnet are all visible to each other. Hence, we assume that all machines within a subnet can directly access the machines that the representative source machine can access. Only a part of the machines outside the subnet are visible from within the subnet, due, e.g., to firewall restrictions. We model the accessibility of machines identified through the scans as direct edges in the network graph. That is, machine m_1 is connected in our model to machine m_2, if m_2 is visible from m_1 or vice versa.

In addition, Nessus reveals for each identified host its operating system. The network contained hosts running Windows and Linux (with a few versions of each operating system). Nessus also identifies softwares with potential vulnerabilities that run on the machines. Our model contains about 50 such software, including well known applications such as *openssh*, *tomcat*, *pcanywhere*, ftp services, and many more.

Nessus finds vulnerabilities of varying importance. For the purpose of this experiment we ignored all the lesser vulnerabilities, which do not allow an attacker control of the system. We remain with about 60 serious types of vulnerabilities that exist in the network. We remove from the network all hosts that do not run any software for which a serious vulnerability exists, remaining with about 23 hosts.

For constructing our attack goal, we took six random hosts from the innermost subnet, and set them as the target hosts. The problem goal is to gain control over one of these six hosts.

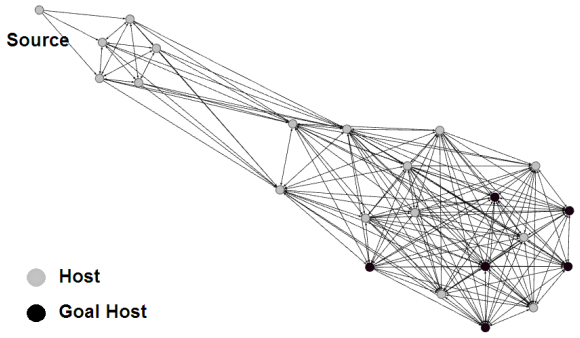


Figure 4: Connectivity graph of a network. Each node is a host computer, a directed edge between two hosts (u, v) means host u can initiate a connection to node v

5 Evaluation

We now compare the utility of various graph centrality measures in ranking the set of possible machine vulnerabilities to be fixed. In this paper we focus on the task of increasing the cost of the minimal attack plan. That is, we use the various metrics computed over the LAG or the planning graph to rank the vulnerabilities to be fixed, and check which ranking induces an increase in the minimal attack plan cost using less fixes.

5.1 Domains

We experiment with a benchmark network from previous research [Hoffmann, 2015] [Durkota *et al.*, 2015], as well as the network of the large organization that we scanned. The network scanned contained 23 hosts including a host representing the Internet. The hosts had 144 critical vulnerabilities which an attacker could leverage. The dataset from the literature - LocalPlus-20 originally contained 23 hosts, we added 3 more hosts and slightly altered it's connectivity graph. In the end the graph contained 26 hosts and 26 vulnerabilities. The statistics about the LAG produced from the above networks are presented in Table.2.

We have searched for additional publicly available networks and found none. We also explored additional simulated benchmarks, but these presented very artificial networks (e.g.

	$\|V\|$	$\|E\|$	Avg. Degree	Diameter
Scanned network	1003	1591	3.172	15
LocalPlus-20	394	560	2.84	16

Table 2: Attack Graphs Statistics

where each machine had only a single vulnerability), and the results over these networks were uninteresting.

5.2 Methods

We compare here the following metrics:

1. Plan count
2. Betweenness
3. PageRank
4. Closeness
5. Random ranking

Plan count is the number of shortest plans in which a vulnerability participates. This is computed using the plan enumeration procedure. Vulnerabilities are ranked by decreasing number of plans in which they participate.

All metrics are computed over both the LAG and the planning graph, except for the plan count, which is computed only over the planning graph. As a node in the LAG can appear multiple times in the planning graph (For instance between no-op actions, e.g: $p \rightarrow noop \rightarrow p$), we count the different appearances of a node in the planning graph.

5.3 Procedure

We performed the experiments in the following manner; For each centrality method we begin with the original graph (LAG or planning graph), and compute the metric for all vulnerability nodes in the graph. We then rank all the nodes according to the centrality method by decreasing value.

We select the node with the highest centrality measure to be fixed first. The vulnerability corresponding to this node is now removed, and we recompute the LAG or the planning graph without this node. Then, we recompute the metric over the new, revised, graph.

In addition, we enumerate the set of attack plans over the original planning graph using Algorithm 1. We identify the subset of plans with the minimal cost (shortest plans). Whenever we remove a vulnerability following the above procedure, we also remove all the shortest plans that use this vulnerability. We end when there are no more shortest plans left.

5.4 Results

Figure 5a presents the reduction in the number of shortest attack plans after every patch (removal of a vulnerability on a specific host) on the simulated Local+20 benchmark. On this network, the only two metrics that supply any useful information are the shortest plan count, and Betweenness over the planning graph. Both methods allow the administrator to remove all shortest attack plans after patching only 4 vulnerabilities. All other centrality metrics do not perform better than a random ranking.

The network of the real organization that we scanned provides completely different results (Figure 5b). On this network, while selecting vulnerabilities that appear in the largest number of shortest plans reduces the amount of plans most rapidly at first, PageRank over the planning graph manages to remove all shortest plans using the minimal number of patches. Closeness over the planning graph and the plan count method remove all attack plans with the same number of patches (13). Over this network, all metrics computed over the planning graph provide better rankings than metrics computed over the LAG.

The difference between the performance of the metrics over the real network and the simulated benchmark clearly present the urgent need for experiments with real world data. Simulated networks in this case may not model properly the real world, and results over them may be misleading.

6 Related Work

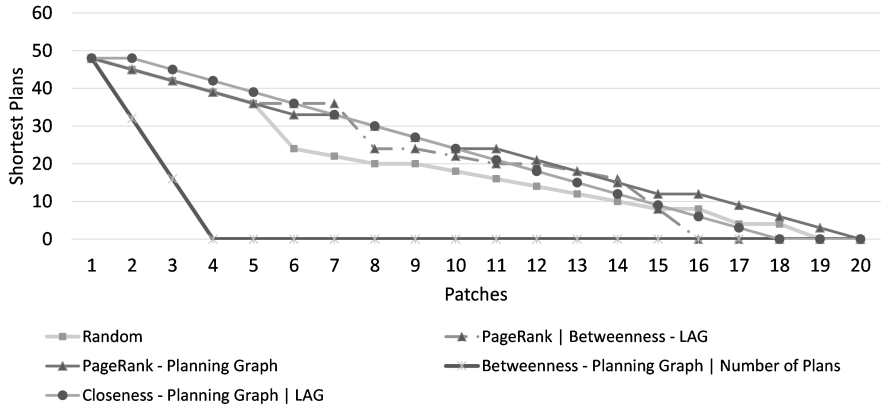
Attack Graphs have been used to depict possible ways for an attacker to compromise a computer network. Almost two decades ago DARPA created attack graph manually as part of red-team analysis. Initially, attack graphs were used to better visualize the paths an attacker can take in the network. Once attack graph could easily be generated automatically, researches have used them to improve the security of the networks. They did so by a number of different methods. We presented works that can help determine what vulnerabilities to patch in an organizational network. Additional works exist [Durkota *et al.*, 2015] to help determine countermeasure placements in the network (like IPS or Honeypots) but they are out of scope for this work.

6.1 Finding Optimal Attack Plans

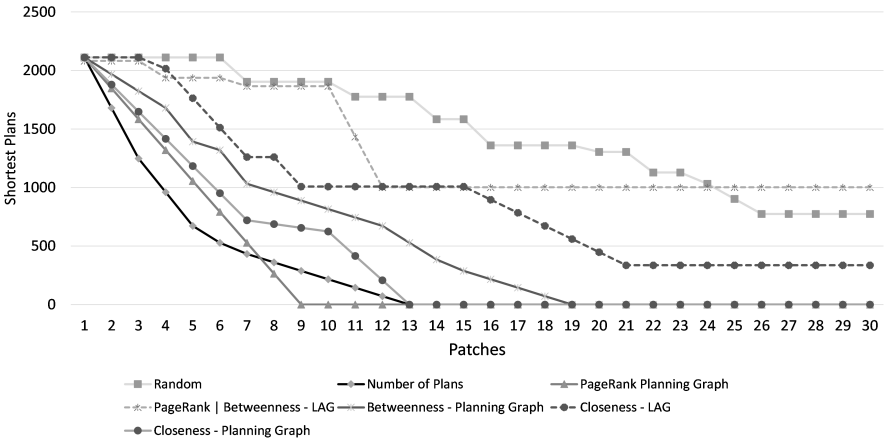
Many researches have assumed some metric on actions in the attack graph [Obes *et al.*, 2013]. The metrics usually represents cost, like the time it takes to launch exploit, risk of detection and so on. Another common metric for actions is the probability of success when performing the action. [Wang *et al.*, 2008] Researchers then tried to find attack plans which minimize/maximize the suggested metric. The assumption is that rational attacker will first try to launch attacks that minimize the cost for the attacker. The downsize of many of those models are simplifying assumptions on the attacker, which are not always realistic in the real world. Example of those assumptions are the fact the attacker needs to know beforehand the structure of the network, or that he cannot change the structure of the network. When trying to relax those assumptions, using Contingency Planning, MDP or POMDP to achieve realistic results, the runtime for deriving conclusions makes it not practical for real-size networks [Hoffmann, 2015] [Shmaryahu, 2016]. More over, test shows [Sommestad and Sandström, 2015] that attack graphs do not always represent all the actual paths an attacker can take. So trying to eliminate numerous plans in respect to some metric might not always prevent an attacker from achieving his goals.

6.2 Denying Access to the Goal

Another common use of attack graphs is finding a set of conditions to patch which will prevent the attacker from reach-



(a) Local+20 network



(b) Large organization network

Figure 5: Amount of shortest attack plans (y axis) available after applying k patches (x axis) to the most central nodes, according to the different centrality methods.

ing the goal. Works in this area use various methods such as minimum-cost SAT solving [Huang *et al.*, 2011] and specialized methods [Albanese *et al.*, 2012] invented for this task. The down-side of this method is that in practice, even after finding minimal set of conditions to patch it may still contain a significant amount of vulnerabilities to patch, which will not be possible without significant IT resources. This leads to our goal to minimize the number of attack plans in the graph that could be used to reach the goal.

6.3 Patching to Minimize Paths to the Goal

In their work [Hong and Kim, 2013] [Hong *et al.*, 2014] propose to use network centrality measures to find which vulnerabilities to patch first. To our understanding, this work proposes using a two-layer graph. The first layer, representing the hosts in the system. A directed edge between two nodes (a, b) means that when controlling node a, an attacker is able to advance to node b (using exploit, or a similar manner). The second layer is an AND-OR tree containing all the ways to compromise a machine from an arbitrary other machine. The authors then compute different network centrality measures on the first layer to find which vulnerabilities to patch. To our understanding, the absence of the knowledge on how to gain access to a host (which is depicted in layer two) when computing centrality measures on the first layer, can often yield sub-optimal results.

7 Conclusion and Future Work

We discuss in this paper metrics for ranking the vulnerabilities to patch in a computer network. We focus on the problem of increasing the cost of the shortest attack plan. We show that metrics computed over the planning graph — a data structure from automated planning — provide much better rankings. In addition, the planning graph allows us to enumerate the set of shortest plans, providing a new ranking metric based on vulnerability appearance in shortest plans.

We experiment with a real world network, which we defined using a scan of a large organization computer network. As such, this is one of the first papers to report results over an attack graph of a real network. We also experimented with a standard simulated benchmark. It is interesting to see that the results over the simulated network are very different than results over the real network, emphasizing the urgent need for

additional real world networks for experiments.

In the future we intend to experiment with additional interesting real world networks. We would also explore additional optimization problems, such as removing all identified attack plans. The plan enumeration procedure that we used has an exponential complexity, and we intend to explore sampling methods to allow for rapid estimation of the set of shortest plans.

References

[Albanese *et al.*, 2012] Massimiliano Albanese, Sushil Jajodia, and Steven Noel. Time-efficient and cost-effective network hardening using attack graphs. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[Beale *et al.*, 2004] Jay Beale, Renaud Deraison, Haroon Meer, Roelof Temmingh, and Charl Van Der Walt. *Nessus network auditing*. Syngress Publishing, 2004.

[Blum and Furst, 1997] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997.

[Brandes, 2001] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[Cukier and Panjwani, 2009] Michel Cukier and Susmit Panjwani. Prioritizing vulnerability remediation by determining attacker-targeted vulnerabilities. *IEEE Security & Privacy*, 7(1):42–48, 2009.

[De Mello and Sanderson, 1990] LS Homem De Mello and Arthur C Sanderson. And/or graph representation of assembly plans. *IEEE Transactions on Robotics and Automation*, 6(2):188–199, 1990.

[Durkota *et al.*, 2015] Karel Durkota, Viliam Lisỳ, Branislav Bošanskỳ, and Christopher Kiekintveld. Optimal network security hardening using attack graph games. In *Proceedings of IJCAI*, pages 7–14, 2015.

[Gefen and Brafman, 2012] Avitan Gefen and Ronen I Brafman. Pruning methods for optimal delete-free planning. In *ICAPS*, 2012.

[Hoffmann, 2015] Jörg Hoffmann. Simulated penetration testing: From" dijkstra" to" turing test++". In *ICAPS*, pages 364–372, 2015.

[Hong and Kim, 2013] Jin B Hong and Dong Seong Kim. Scalable security model generation and analysis using k-importance measures. In *International Conference on Security and Privacy in Communication Systems*, pages 270–287. Springer, 2013.

[Hong *et al.*, 2014] Jin B Hong, Dong Seong Kim, and Abdelkrim Haqiq. What vulnerability do we need to patch first? In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 684–689. IEEE, 2014.

[Huang *et al.*, 2011] Heqing Huang, Su Zhang, Xinming Ou, Atul Prakash, and Karem Sakallah. Distilling critical attack graph surface iteratively through minimum-cost sat solving. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2011.

[Morrow, 2012] Bill Morrow. Byod security challenges: control and protect your most sensitive data. *Network Security*, 2012(12):5–8, 2012.

[Obes *et al.*, 2013] Jorge Lucangeli Obes, Carlos Sarraute, and Gerardo Richarte. Attack planning in the real world. *arXiv preprint arXiv:1306.4044*, 2013.

[Ou *et al.*, 2005] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. Mulval: A logic-based network security analyzer. In *USENIX security*, 2005.

[Ou *et al.*, 2006] Xinming Ou, Wayne F Boyer, and Miles A McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345. ACM, 2006.

[Page *et al.*, 1999] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[Sarma *et al.*, 2015] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. Fast distributed pagerank computation. *Theoretical Computer Science*, 561:113–121, 2015.

[Sawilla and Ou, 2008] Reginald E Sawilla and Xinming Ou. Identifying critical attack assets in dependency attack graphs. In *European Symposium on Research in Computer Security*, pages 18–34. Springer, 2008.

[Shmaryahu, 2016] Dorin Shmaryahu. Constructing plan trees for simulated penetration testing. In *The 26th International Conference on Automated Planning and Scheduling*, page 121, 2016.

[Shostack, 2003] Adam Shostack. Quantifying patch management. *Secure Business Quarterly*, 3(2):1–4, 2003.

[Sommestad and Sandström, 2015] Teodor Sommestad and Fredrik Sandström. An empirical test of the accuracy of an attack graph analysis tool. *Information & Computer Security*, 23(5):516–531, 2015.

[Wang *et al.*, 2008] Lingyu Wang, Tania Islam, Tao Long, Anoop Singhal, and Sushil Jajodia. An attack graph-based probabilistic security metric. In *Data and applications security XXII*, pages 283–296. Springer, 2008.

[Wang, 2006] David Eppstein Joseph Wang. Fast approximation of centrality. *Graph Algorithms and Applications 5*, 5:39, 2006.

[Zhang *et al.*, 2014] Su Zhang, Xinwen Zhang, and Xinming Ou. 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*, pages 317–328. ACM, 2014.