Designing Scalable and Effective Decision Support for Mitigating Attacks in Large Enterprise Networks

Zhiyun Qian[1], Z. Morley Mao[1], Ammar Rayes[2], and David Jaffe[2]

[1] University of Michigan, Ann Arbor, MI 48105, USA
{zhiyunq,zmao}@umich.edu
[2] Cisco Systems, Inc. San Jose, CA 95134, USA
{rayes,djaffe}@cisco.com

Abstract. Managing numerous security vulnerabilities has long been a difficult and daunting task especially due to the complexity, heterogeneity, and various operational constraints of the network. In this paper, we focus on the task of mitigating and managing network-device-specific vulnerabilities automatically and intelligently. We achieve the goal by a scalable, interactive, topology-aware framework that can provide mitigation actions at selectively chosen devices. The intuition behind our work is that more and more network devices are becoming security-capable so that they can be collectively used to achieve security goals while satisfying certain network policies.

The intelligence utilizes integer programming to optimize a quantifiable objective conforming to the policy of a given network. An example would be to find the minimum number of network devices to install filters to effectively protect the entire network against potential attacks from external untrusted sources. The constraints of the integer programming are mainly based on the network topology and settings of vulnerable devices and untrusted sources. Our novel implementation uses an iterative algorithm to scale to networks of tens of thousands of nodes, and we demonstrate the effectiveness of our framework using both synthetic and realistic network topologies. Besides scalability, our tool is also operationally easy to use by enabling interactivity to input additional constraints during runtime.

Keywords: vulnerability management, optimization, integer programming.

1 Introduction

With the increasing complexity of the Internet, enterprise networks have grown in both size and complexity, so have associated network devices which not only perform packet routing and forwarding but are also equipped with network management and security functionalities such as packet filtering. These devices can act as firewalls to partition the network into distinct groups and prevent intrusions by filtering unwanted traffic based on attributes such as source/destination IP address, source/destination port, TTL values, *etc.* These can provide intermediate or temporary solutions to defend the network, for instance, by limiting access to potentially vulnerable services only to trusted/valid IPs through the use of ACLs (Access Control List).

Given the broad range of security vulnerabilities in existing networks ranging from buffer overflow, code injection [1] to denial of service [2], it may not be sufficient to rely on simple firewalls. However, many of such vulnerabilities can be mitigated at the network level due to significant advance in network security technology manifested in devices such as Network Intrusion Detection System (NIDS) and Network Intrusion Prevention System (NIPS).

If a network device, *e.g.,* Cisco Intrusion Prevention System (IPS) device [3], has advanced Deep Packet Inspection (DPI) capability, packet filters can be set up based on payload. They are capable of detecting and preventing a variety of intrusions. For example, the *DNS Implementations Insufficient Entropy Vulnerability* can be mitigated by installing a signature on the DPI-capable device to detect a DNS flood possibly leading to DNS cache poisoning, reflection, or amplification attacks [4].

Note that network level defense suffers from the shortcoming by assuming where attacks can enter the network. Thus our proposed framework shares the same assumption, revealing the difficulties of fully defending against internal attacks. Nevertheless, network level defense complements well other types of defense such as host-based intrusion detection system. The alternative of applying a patch to fully fix the vulnerability may not be immediately adopted because of several reasons. First, a patch for the vulnerable software may not be available. Second, the patch may not be fully tested and may introduce unwanted side-effects. Finally, applying the patch may require rebooting the device, introducing network disruption. Since the basic firewall capability is built-in for virtually every modern router and switch (*e.g.,* Access Control List), various choices with different tradeoffs exist in terms of how to temporarily protect the network.

For those vulnerabilities that cannot be prevented at the network level, applying the patch directly to the vulnerable software is preferred since patching only incurs one-time overhead and provides the best protection. However, considering the number of devices in the network that are potentially very diverse (as shown in the next section), knowing what to patch first without causing much disruption can be very challenging, let alone consider the case when the options of patching vs. network-level defense are both available. Finding the best strategy considering various tradeoffs can be a daunting task. For that purpose, we have developed *a framework using integer programming that considers various tradeoffs and makes optimal suggestions on which routers to reconfigure/patch to prevent intrusions based on the topology of the network and policies/preference of network/sys admin*. In what follows, we will use the term *filter* as a general term for network-level defense.

Our work is quite applicable as large networks today often deploy DPI capable security systems not only at a few external gateways but also internally to defend against internal threats. Furthermore, it is the trend that more network devices will have such security capabilities built-in. There is however no prior work to thoroughly analyze how to plan or utilize these resources wisely. More specifically, decision has to be made to determine which devices and what operations are to be performed to address known vulnerabilities while minimizing overhead without compromising security protection. The overhead includes management complexity, as well as performance penalty introduced by the size of DPI signatures or firewall rules [5].

We develop a prototype framework to help network/sys admin manage security vulnerabilities at the network level by integrating two main primitive operations – filter and patch. Our novel iterative implementation allows the system to easily scale to networks of thousands of network devices. Furthermore, we build operational interactivity into the design to facilitate constraint modification during run time. As with any model-based approach, the guarantees offered depend on the model accuracy. Despite the simplicity of the abstraction used in our model, it is sufficient for our purpose as shown later. Furthermore, our approach has the benefit of being independent of low-level implementations, *e.g.,* how to configure the filtering rules. Our framework also complements existing work in formal analysis [6] to ensure the correctness of rule configurations.

The paper is organized as follows. §2 motivates our work by revealing the heterogeneity and complexity of real networks. §3 introduces our framework. §4 then focuses on how we translate the security management problem into an optimization problem illustrated using a simple example. We evaluate our tool against several real networks to demonstrate its effectiveness in §6. §7 describes several related work. Finally we conclude with discussions in §8 and §9.

2 Network Device Diversity in Real Networks

To motivate the need for a framework to deal with complex network goals and constraints, we first want to understand how diverse real networks are. We leverage the inventory data from Cisco's remote router management system (formally known as Cisco Inventory and Reporting or IR [7]). In a nutshell, Cisco IR allows Cisco to remotely manage the network of a company that chooses to use the service (many big companies from different industrials use the service).

Interestingly, from these real networks, we found there are many different versions of operating systems running on their network devices (*e.g.,* shown in Figure 1). The Y-axis is $\frac{\#\ of\ different\ IOS\ version}{\#\ of\ devices/chassis}$ which indicates the degree of variety of the network devices. The X-axis is different organizations whose networks are managed by Cisco's Inventory Reporting application. The number of devices for each of the organizations range from hundreds to a thousand. Surprisingly, the most diverse network has more than 180 different OS versions. This many different OS versions cause complex many-to-many relationships between OS versions and corresponding vulnerabilities as shown in Table 1, securing the entire network taking into account all OS versions and device vulnerabilities in an optimized fashion is quite challenging. Furthermore, some of the vulnerability may be more critical than others, some incur more overhead (*e.g.,* downtime). The surprisingly diverse and complex network devices motivate the need to mitigate and manage their vulnerabilities automatically and intelligently. To ensure practical relevance, we design our framework to handle multiple vulnerabilities, allowing users to specify these in a quantifiable metric.

3 The Framework

In this section, we first describe the high-level framework and the building blocks to support our objective of providing intelligent attack mitigation decision support. As example mitigation support of interest to network/sys admin could be "finding the minimum number of network devices to install filters to prevent attack X". This work is based on the observation that many of the security management problems can be

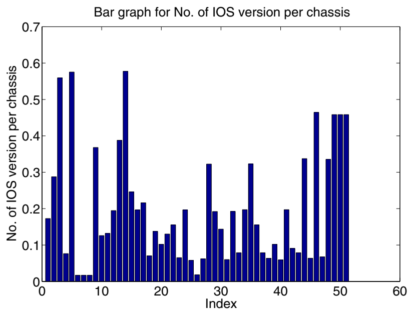

Fig. 1. The number of unique OS version per chassis in different real networks

Table 1. An example of multiple vulnerabilities on various versions of Cisco IOS

Vulnerability ID / IOS version	1	2	3	4
11.0(11)BT			x	x
12.0(10)ST	x	x	x	x
12.0(11)S4		x	x	

modeled as optimization problems. We present our simple and elegant method based on integer programming to help solve this class of problems.

Our framework is designed to be built on top of existing network information including network topology, configuration files of the network devices, the security alert data and the network/sys admin's objective and requirements. We describe the inputs below, also illustrated in Figure 2.

Inventory and vulnerability information contains data such as device type and running services (including PCs and routers/switches). IT departments in companies often track a subset, if not all, of such information already. For instance, Cisco offers remote router management that tracks the inventory and vulnerability information of all the routers. The information can be automatically collected using both open source and commercial tools [8, 9]. As an open standard, Open Vulnerability Assessment Language (OVAL) [9] is an XML-based language for specifying machine configuration tests. OVAL-compatible scanners can be used to gather vulnerability information of the devices given OVAL definitions. For network devices, the network/sys admin typically runs the scanner via SNMP to collect the device info as well as the OS version and its patch level. We ran the similar test on our local network which has several hundreds of network devices and it takes only less than a minute to finish.

Security alerts contain vulnerability information for software on different platforms (both PCs and routers/switches) and provides the basic prevention or detection recommendations. For example, the alerts may disclose whether a patch is available for a particular piece of software. Such information or service is published by various vendors such as Cisco Intellishield [10]. For instance, we can easily tell, according to the security alert service, that the *Multiple SNMPv3 Implementations Hash-Based Message Authentication Code Manipulation Vulnerability* can be mitigated by either applying patches, configuring ACLs, or installing IPS signatures on DPI-capable devices.

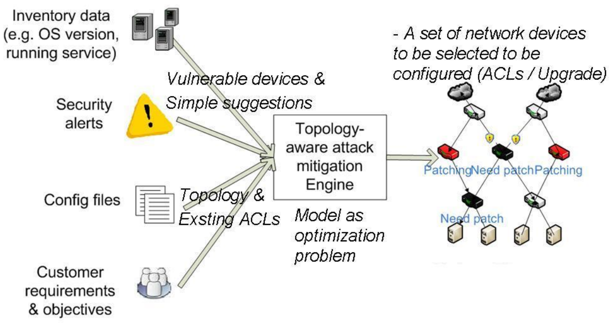
Inventory data (e.g. OS version, running service)
Security alerts
Config files
Customer requirements & objectives
Vulnerable devices & Simple suggestions
Topology & Exsting ACLs
Topology-aware attack mitigation Engine
Model as optimization problem
- A set of network devices to be selected to be configured (ACLs / Upgrade)
Patching
Need patch
Patching
Need patch

Fig. 2. Our framework for making attack mitigation suggestions

Network Topology. Typically, this information is maintained by IT department already. If not, there are techniques to reconstruct layer-three topology based on their IP addresses [11] from router configuration files. The topology information can also be obtained by probing the network [8, 11], typically, by real network management tools such as NetMRI [12].

Objective is used to describe the network-dependent properties that can either be specified by the network/sys admin or inferred automatically discussed later.

Here we assume that different kinds of attack mitigation building blocks can be used on each network device depending on its unique capability:

- Configure the ACL (Access Control List) to guard against certain (untrusted) IP range and/or ports.
- Configure the firewall to stop unwanted traffic.
- Install an appropriate packet filter based on signatures for identifying malicious payload if the device is IDS or IPS-enabled.
- Apply the patch on the devices or the end-hosts.
- Other network device built-in capabilities such as IP Source Guard enabled on many Cisco devices.

4 Problem Formulation - Optimization

From the input of the framework, we can extract the network settings, the vulnerable nodes (PCs or routers/switches), and more importantly, the goals and constraints. For example, network/sys admin may want to balance the number of filtering rules on a particular router (due to processing overhead) and the overall number of interfaces to be reconfigured (due to management overhead). The constraint can be, for example, to protect all of vulnerable nodes or to protect only nodes with the most severe vulnerability. Based on the problem requirement, it is natural to cast it as an optimization problem which we can model using integer programming. The reason for this choice is that integer programming is not only very simple and intuitive to use, but also provides a small and well-defined interface, thus allowing various Integer Programming Solvers to be optimized separately. We will illustrate how these variables are defined and how to use different objective functions and constraints to solve several types of realistic security management problems.

Note that our framework aims to provide intelligent suggestions for various security management problems. More specifically, the framework supports filtering and patching decisions based on various constraints/tradeoffs for multiple vulnerabilities.

4.1 Overview

Variables. For each interface in the network, we define a binary integer variable x_i, which can either be 0 or 1 indicating whether this interface is configured with a filter (for normal switch/router) or a signature (for NIDS/NIPS). Alternatively, a variable can be defined for each node (PC or switch/router) rather than an interface indicating whether a node has filters installed (regardless of the interfaces). Similarly, for each node, we define a binary integer variable y_i which indicates whether this particular node is to be patched.

Note here we can omit a variable or always assign the variable to zero if a network device or interface does not support the basic mitigation support (*e.g.,* an older version of router without ACL support). To address multiple vulnerabilities, we define different sets of variables $x_i^{(k)}$, $x_{i+1}^{(k)}$ *etc.* for the k_{th} vulnerability. In comparison, we also define a special patch variable y_i. Since patching one node usually eliminate all the vulnerabilities under consideration, either all k vulnerabilities are protected by filters or the node is patched suffices the security requirement. In the following discussions, any variables defined will be a binary integer variable unless otherwise specified.

Objective function can express many different goals but with the limitation that it has to be linear function of the variables of the form $\sum_i a_i x_i$. Despite this apparent limitation, it is sufficient to solve many of the security management problems. For example, the objective function could be $\sum_i x_i$ which is the total number of interfaces that are configured to install filters or NIDS/NIPS signatures. The goal would be to minimize this value.

Constraint is of the form $\sum_i a_i x_i <= b$ where a_i and b are constants. A sample constraint would be defined as $x_1 + x_2 + x_3 + y_1 >= 1$ where x_1 is an untrusted interface and x_3 is an interface that belongs to a vulnerable device y_1. This constraint means that there has to be at least one filter along this path to protect the vulnerable device or the device can be patched by assigning variable y_1 to 1. If there is no patch available yet for the vulnerability or due to other business reasons (*e.g.,* downtime), we can simply remove the variable y_1.

4.2 An Example

A simple example that illustrates how integer programming can be set up is shown in Figure 3. We do not consider patch in this example for simplicity. The topology consists of a set of routers (from x_1 to x_7) and a set of servers that are vulnerable to a newly discovered vulnerability in an enterprise network. Assuming that the operator prefers not to simply patch these servers due to reasons such as possible downtime to their customers, so we remove all the patch variables y_i. The alternative is to install a corresponding signature for this vulnerability to filter malicious incoming packets on the routers (or any other mitigation building blocks such as ACLs), assuming the signature is available. The question is where to install such filters. A simple solution would be to

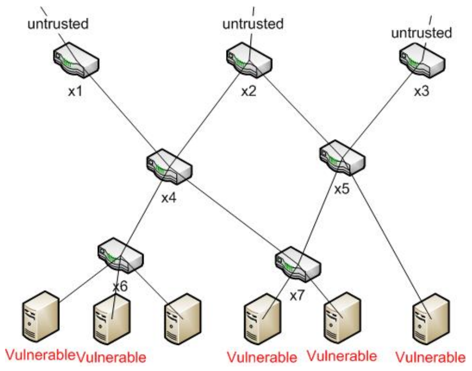

Fig. 3. Example 1 - topology

install it on every gateway (x_1, x_2 and x_3), but it is not an optimal solution in terms of the number of devices involved (assuming a desirable goal is minimal complexity).

A better strategy is to install the filters on x_4 and x_5 only. This optimal solution can be found by solving the corresponding integer programming problem that is translated from the current network setting (network topology, untrusted source interfaces and vulnerable nodes). Below are the definitions of objective functions and constraints for this example.

Objective Function. Since we are trying to minimize the number of nodes that are installed with filters, the objective function is defined as $\sum_{i=1}^{7} x_i$.

Constraints:

$x_1 + x_4 + x_6 >= 1$
$x_1 + x_4 + x_7 >= 1$
$x_2 + x_4 + x_6 >= 1$
$x_2 + x_4 + x_7 >= 1$
$x_2 + x_5 >= 1$
$x_3 + x_5 >= 1$
$x_i >= 0$ for each $1 <= i <= 7$

We can easily get the answer from this integer programming setup: $x_4 = x_5 = 1$, $x_i = 0$ for $i \neq 4$ and $i \neq 5$. Sometimes, however, the number of filters on x_4 and x_5 may be too large so that the network/sys admin may want to avoid using them. This can either be solved by setting a different objective function (§4.3) or allow the user to interact with the tool and provide feedback to the tool (§6.3).

4.3 Objectives

Network/sys admins may specify different kinds of objective functions that they want to optimize based on a given set of constraints. Here we describe some common objective functions of interest:

Minimal involvement - minimum number of network device configuration changes. The objective function is defined as $\sum_i x_i$ where x_i is the variable for each node indicating whether a particular node has been configured for filters as discussed before.

Note here once the node is configured, then it can be applied to any number of interfaces on that device without additional cost in our formulation. The reason for this policy is that network operators may want to involve smallest number of devices to defend their network for simplicity or management overhead considerations.

Minimal management complexity - minimum amount of management complexity imposed. The objective function is defined as $\sum_i (((n_i + 1)^2 - {n_i}^2) \times x_i)$ where $(n_i + 1)^2 - {n_i}^2$ is the amount of management complexity increased by adding a new ACL entry on an interface, n_i as the number of ACL entries for the corresponding interface and n_i^2 is the management complexity of a given interface where n is the number of entries of ACLs configured. The incentive for this policy is that due to complex ACL matching rules, a large number of ACL entries are known to be difficult to manage.

Minimal number of devices involved - minimum number of devices that are either to be configured for filters or patches. The objective function is defined as $\sum_{i,j} x_i^{(j)} + \alpha \sum_m y_m$, where $x_i^{(j)}$ is the variable for node i and vulnerability j indicating whether this node has been configured for filters to prevent vulnerability j, y_m is the variable for node m indicating whether this node is to be patched (multiple real patches for different vulnerabilities are combined into this single variable). α is the constant coefficient which balances the choice between installing filtering and patching. Normally it is larger than the cost of installing filters. However, as previously stated, if patching one node can eliminate the need for filters on many nodes, then it may be a preferred choice. This is the case given multiple vulnerabilities in one or more nodes, patching them obviates any other filters. In fact, modern routers tend to have multiple vulnerabilities due to their complexities [13]. §2 describes how to set up the constraints for multiple vulnerabilities and patch operation. We can also define the objective function in terms of interfaces instead of nodes.

Minimal network performance overhead - minimize possible throughput and latency performance overhead imposed by installing filters. The idea is that although most network devices support ACL or firewall rules, they come with a cost. Even for modern devices where hardware support has been widely applied to optimize the ACL or firewall rules, for example, by using Content-addressable memory (CAM), the throughput can drop significantly [14] when the number of ruleset exceeds certain threshold (depending on vendors and models). The same also applies to DPI devices. As a result, the objective function can be defined as $\sum_i k_i$ where k_i is defined based on the number of existing filters (denoted by n_i) on interface i. $k_i = 0$ when $n_i <= s$ and $k_i = {x_i}^j + n_i - s$ when $n_i > s$.

Intuitively, the objective function captures the performance penalty imposed on each interface due to filters and the overall impact. Note that $k_i = 0$ when $n_i <= s$ is approximated because s is relatively larger than the number of filters to be placed on a single interface. Typical s for modern routers is in the order of hundreds. An alternative objective function would be to minimize $\max(k_i)$ because usually the overall network performance is determined by the bottleneck component. This policy is to help eliminate the scenario where filters are installed only on few core routers which may deeply impact the network performance.

Note that these objective functions can be combined to achieve a balance between different goals. Here in many cases the cost of placing filter is to be set identically for simplicity. However, we do offer some simple heuristics on how the cost can be selected. For example, a network device with high capability and low overhead for installing filters should generally be considered low cost. Another example is that when the number of existing filters on the device is already large, it should be considered high cost. Further, we allow the users to tune the result in an interactive fashion which provides much better usability as shown in § 6.3.

4.4 Constraints

Below are some examples of useful constraints.

Installing Filters to Protect Vulnerable Nodes. For each vulnerable node j and untrusted node i, enumerate all possible paths from i to j. For each path, consider the constraint $x_i + .. + x_j >= 1$ where each variable can be the variable for the node or the interface, depending on the problem setup. If this constraint is satisfied, then a vulnerable node is guaranteed to be protected on this particular path (since at least one interface/router along the way will be configured to filter malicious packets). Similarly, we can apply this for every vulnerable node and untrusted node pair to ensure global safety. There are variants where one can specify the constraint to be $x_i + .. + x_j >= 2$ to increase defense redundancy.

Filters or Patch. Given a particular vulnerability for which a patch is available, a vulnerable node j and an untrusted node i, enumerate all possible paths from i to j. For each path, consider the constraint $x_i + .. + x_j + y_j >= 1$ where x_i to x_j can be the variables for the node or the interface. y_j is the additional variable (defined in objective functions) indicating whether this node will be patched. This constraint will be satisfied either when there is a filter along the path or it is patched. Note that in practice, we might need several different patches to be installed for diverse vulnerabilities, but generally we consider them logically as one aggregate patch in our abstraction. Exceptions are made when some vulnerabilities have corresponding patches but some do not. We can also support this case by partitioning the vulnerabilities into patchable ones and unpatchable ones, as discussed in §4.3 and §4.4.

Latency Constraint. For simplicity, we can model the latency constraint using filtering rules. Intuitively, with more rules, the router needs to spend more time processing them. For a beginning node i and an ending node j on a path, consider the constraint $x_i^{(1)} + .. + x_j^{(1)} + x_i^{(2)} + ... + x_j^{(2)} + ... + x_i^{(n)} + ... + x_j^{(n)} <= c$, where each $x_l^{(k)}$ is the variable defined for each interface along the path, assuming that $x_l^{(k)} = 1$ is equivalent of adding one filtering rule on an interface. c is a constant describing the maximum number of increased filtering rules allowed. $x_i^{(k)} + ... + x_j^{(k)}$ is the number of filtering rules added for k_{th} vulnerability along the path from node i to node j. Obviously, $\sum_k x_i^{(k)} + ... + x_j^{(k)}$ is the overall filtering rules added for the path.

5 Implementation

The Integer Programming Solver we use is CPLEX-11.0 [15]. We first implement our tool in a brute-force, naive manner, by calculating all possible constraints through the

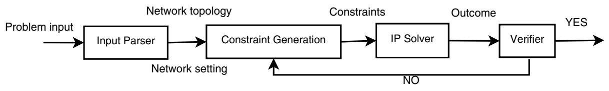

Fig. 4. Logic flow of iterative implementation

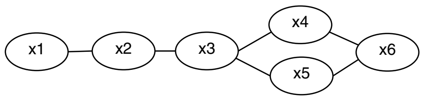

Fig. 5. An example topology that shares common path

enumeration of all paths between untrusted node and vulnerable node. The problem is that when the graph is dense enough, the number of paths between two nodes could be exponential with respect to the number of nodes. We may argue that most real topologies are usually not dense graphs, but many large networks usually have redundant links/backup nodes to provide availability and failure resilience. To address this problem, we have proposed the novel implementation that uses *an iterative approach* to incrementally add constraints to reduce the search space for all possible paths between two nodes. Further, the iterative implementation produces the same optimal result as the naive implementation.

Formally, our problem is $min\ c^T x$, under a set of constraints I. Note that the size of I can be very large. We propose to iteratively add a subset of I and generate a temporary result for the subset of constraints. The hope is that the outcome computed based on the subset of I will satisfy the ultimate constraint that all of the vulnerable nodes are protected before all of the constraints in I are added. It is illustrated in Figure 4.

This approach is based on the following observations:

1. We may not need all the constraints in I to compute the optimal solution because there are many redundant constraints. It is unnecessary to go through all of them. For example, $x_1 + x_2 + x_3 >= 1$ is redundant if there is a constraint $x_1 + x_2 >= 1$. These cases should be handled automatically by standard linear programming or integer programming solver. However, there are many other constraints that can share common variables while neither one of them is redundant. See Figure 5 as an example, there are two paths from x_1 to x_6 whose corresponding constraints look like $x_1 + x_2 + x_3 + x_4 + x_6 >= 1$ and $x_1 + x_2 + x_3 + x_5 + x_6 >= 1$. They share four common variables. It is highly likely, although not always the case, that one satisfied constraint will lead to others being satisfied as well. In real networks, it is not uncommon that several paths share common devices or links. By iteratively adding constraints (in a certain order), we are able to take advantage of such properties.

2. It is relatively easy to verify whether a given set of filters and patch operations will protect all vulnerable devices. This allows us to quickly iterate several times. To check if all vulnerable devices are protected, we perform a breadth-first search in the graph from the untrusted nodes to the vulnerable nodes. The search stops when it encounters a filter or the reached vulnerable node on the edge will be patched.

3. The ordering of added constraints can be determined relatively easily – first add the ones that are less likely to be redundant. Specifically, we pick those shortest attack paths to be the constraints. In general, fewer variables result in less redundancy. If a constraint

Algorithm 1. The iterative algorithm

```
**Initialization**: $I' = \{\}$,
        filter set $F_0 = \{\}$,
        patch set $P_0 = \{\}$,
        objective function $f$.
**repeat** {iteration $i$ from 0 to ...}
    1. Given $F_i$ and $P_i$, compute the set of shortest attack paths and its corresponding
    $I_i$ based on the topology.
    2. $I' = I' \cup I_i$.
    3. Run the IP solver for objective function $f$ under constraints $I'$, get the solution
    $F_{i+1}$ and $P_{i+1}$.
**until** $F_{i+1}$ and $P_{i+1}$ protects all vulnerable nodes
```

with fewer variables is satisfied, the constraints with more variables that share common variables are also likely satisfied.

Formally, the algorithm works as shown in Algorithm 1. It is easy to see that when we select a set of constraints, it limits the search space of the IP solver. The complete set of constraints I will produce the smallest search space. Given a subset of I, we essentially enlarge the search space for the IP solver.

We illustrate the iterative algorithm in Figure 6. The oval here represents the search space of corresponding constraint set. Initially, the search space of the constraint set I'' is generated for the first iteration and then I' is generated in the second iteration. Suppose the initial search space by I'' is too large and causes an incorrect solution (*i.e.,* some nodes will not be protected), while the search space by I' is smaller and the solution can be found within the same range, then there is no need to go to the next iteration and use constraints I to re-compute. The reasoning is that if we found a minimum value in a larger search space (suppose the objective is to minimize), it is guaranteed that we can only find the same minimum or bigger value in a smaller search space too. Since we also check if the result in larger search space satisfies all the constraints, a satisfying result can guarantee that the same minimum value can be found in the final smaller search space.

Note that we are able to approach a good subset quickly and wisely by adding the constraints that are represented by shortest attack path in each iteration. It is essentially an optimistic method by assuming a smaller number of constraints are needed to find the optimal solution which in reality is often the case. By reducing the exponentially large number of constraints, the execution time is significantly improved shown in §6 where most cases take 2 to 5 iterations only.

5.1 Correctness Verification

Note that by the above reasoning, the iterative algorithm is equivalent to the naive approach. To further verify the correctness of our implementation, we ran more than 100 tests up to hundreds of nodes to check that the results generated by naive implementation indeed matches the results generated by the iterative algorithm.

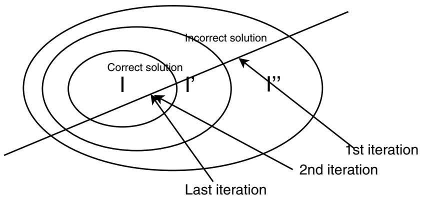

Fig. 6. Illustrating why results computed under a subset of the complete constraint set I are the same as the one under I

6 Evaluation

We describe the evaluation of our framework using both realistic and synthetic network data.

6.1 Real Network Based Evaluation

We have evaluated our tool for a small real network, as shown in Figure 7. The problem setting is as follows (based on a real topology and vulnerabilities): In this network, each node is a router. Node 15 - 18 and 19 - 22 are the untrusted nodes (For simplicity, we do not consider internal nodes as potentially untrusted), and nodes 1 and 2 are the vulnerable nodes. These two vulnerable nodes are installed with different OS versions on the router with a different set of vulnerabilities. Node 1 has vulnerability 1 while Node 2 has vulnerability 1 and 2. All of the vulnerabilities can either be patched or temporarily protected by installing filters. The cost of patch operation is set to be 3 here. The variables are defined in terms of the interfaces visible in the figure.

Our first attempt to set up the problem is to only consider installing filters. Thus the objective function can be setup as:

$$\sum_i x_i^{(1)} + \sum_i x_i^{(2)}$$

where two vulnerabilities are considered together in the objective function.

Alternatively, we can examine each vulnerability independently. These two approaches yield the same solution since the variables in different set of constraints for each vulnerability happen to be disjoint. We first consider vulnerability 1.

The goal is to minimize the objective function defined as $\sum_i x_i^{(1)}$. The constraints are to protect every possible attack path and the solution would be 3 according to the integer programming solver which means only three interfaces need to be configured for filters. Similarly we can obtain the solution for vulnerability 2, which is 2. So it takes $3 + 2 = 5$ interfaces to be configured in order to protect from all of the vulnerabilities.

Our next step is to set up the problem by allowing patch operation, and the objective function is slightly tuned to include the patch variables for the two nodes:

$\sum_i x_i^{(1)} + \sum_i x_i^{(2)} + 2 \times (y_1 + y_2)$

The $2 \times (y_1 + y_2)$ is added to include the cost of patching vulnerable nodes. y_1 and y_2 indicate whether Node 1 and 2 will be patched respectively.

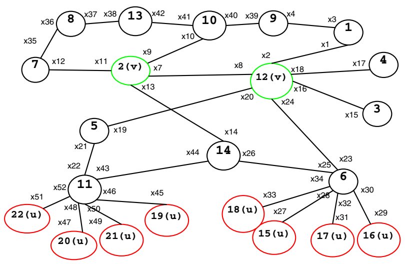

Fig. 7. A small real network for evaluation

The constraints are similar as before, namely to protect every possible attack paths. The difference is that the patch variable y_1 and y_2 are added respectively into each previous constraint depending on the destination node. For example, y_1 will be added to the original constraint $x_{33} + x_{34} + x_{23} + x_{24} + x_2 + x_1 >= 1$ such that $x_{33} + x_{34} + x_{23} + x_{24} + x_2 + x_1 + y_1 >= 1$ forms a new constraint. Since x_4 belongs to Node 1, this means that if the vulnerable node is patched, all the constraints associated with protecting this node can be automatically satisfied.

We obtain the value 4 as the optimal solution where $x_4^{(1)} = x_3^{(1)} = x_5^{(2)} = y_2 = 1$ with every other variable equals to zero.

6.2 Simulation-Based Evaluation

To illustrate the performance of our tool, we simulate various random topologies using the transit-stub model in GT-ITM [16] and randomly select malicious nodes and vulnerable nodes for the problem setup.

In the simulation, we first measure the average running time of our tool against various topologies using our iterative implementation compared with the naive implementation. Then, we measure the number of paths generated and compare with that of the naive implementation. The parameters can be found in Table 2 and Table 3. The sizes of the topologies are approximately 100, 500, 1000, 3000, 5000, 7000 and 10000 respectively.

It can be seen from Figure 8 that the running time (average for ten runs) for naive implementation increases much more quickly with network size compared with the iterative approach. We also verified that they indeed produce the same optimal value. It is quite evident that our iterative approach scales very well. Similarly, Figure 9 shows the overall number of paths for the naive implementation is much larger. This clearly implies much information in the complete constraint set I is quite redundant.

We also illustrate how performance changes when the problem becomes more complex (*e.g.,* with increasing number of untrusted devices and vulnerabilities). We fix a topology with 200 nodes and set up the problem so that the number of untrusted nodes grows together with the number of vulnerable nodes and the types of vulnerabilities. We execute our tool 10 times to measure the average running time and the number of paths/constraints generated. In Figure 10, we can see that our tool can efficiently handle networks of large size.

Table 2. Parameter in the topology generation

Parameters	Variable	Values
# stubs domains per trans node	$F_{s/t}$	4,4,5,7,8,10
# of transit domains	N_t	4,5,6,8,8,8
# of nodes in each transit domain	n_t	5,8,10,10,10,11
Edge prob. between transit nodes	P_t	0.6
# of nodes in each stub domain	N_s	6,6,10,10,11,11
Edge prob. between stub nodes	P_s	0.42

Table 3. Parameter in the problem setup

Parameters	Variable	Value
# of untrusted/malicious node	N_u	10
# of vulnerable node	N_v	10
# of vulnerability	V	3

6.3 Enabling User Interactivity

From the large simulation result, we know that the execution time increases with the problem size (*i.e.,* network size, the number of untrusted/vulnerable nodes, and the number of vulnerability). To understand the bottleneck of the iterative algorithm, we compare the time spent on calculating constraints vs. that on the solver, and observe that the former consumes more than 90% of the execution time. This leads us to develop the heuristic of reusing already calculated constraints. One of the interesting applications it enables is allowing network/sys admin to modify the constraint after he/she sees the result. This effectively turns the tool into an *interactive* one, which is very useful in operational settings. Although theoretically the result computed is the global optimal in terms of the objective function and constraints, the network/sys admin may not have given sufficient input to the tool initially. So allowing changes to the initial result in an

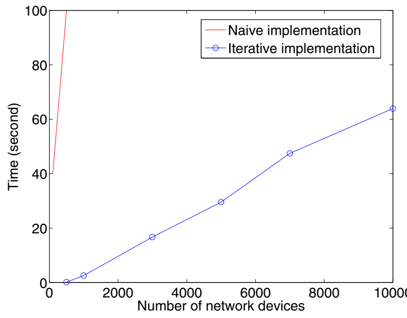

Fig. 8. Execution time for networks of different sizes

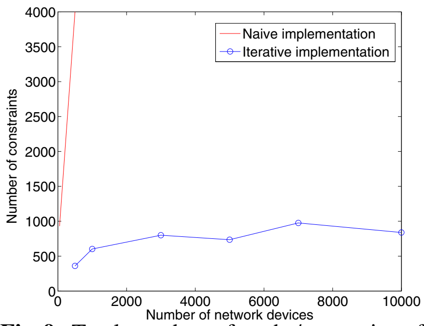

Fig. 9. Total number of paths/constraints for networks of different sizes

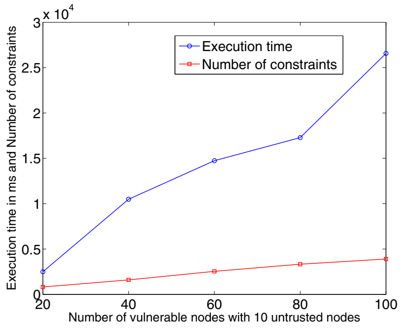

Fig. 10. Number of vulnerable nodes vs. Execution time and number of paths/constraints for size of 1000 network devices

interactive fashion is useful to further tune based on the network/sys admin's domain knowledge of the network. For example, the network/sys admin may want to manually tune the result slightly (*e.g.,* remove filters from some network devices and/or give preference to other devices).

We implemented two types of primitives to allow interactive changes and evaluate their performance. The first primitive is removing a filter assigned on an interface, and the second one is giving preference to a network device for installing filters. The implementation of the first one is straightforward – adding another constraint $\sum_j x_i^{(j)} == 0$ where $x_i^{(j)}$ is the variable indicating whether there should be a filter for vulnerability j on interface x_i. The implementation of second primitive is also simple, *i.e.,* reducing the cost of installing a filter on the specified network device in the objective function (*e.g.,* halving the cost). Given such simplicity, the performance overhead is minimal for supporting interactivity.

7 Related Work

There is a significant amount of research focusing on describing, analyzing and verifying firewall rules [17, 18, 19, 6] to achieve specific global policy. Work on developing a higher level language to describe the firewall rules can be useful, but orthogonal to our work. Investigating issues after the rules are set is complementary to our goal of designing the rules in advance.

Several related work tries to enforce the global policy by distributing policies at different places in the network. An extreme is to distribute the policy to end-hosts instead of to network nodes [20, 21]. This method is topology-ignorant and can be easy to deploy since end-host is easier to change. However, if every policy is to be checked at the end-host (for each packet), it could incur non-trivial overhead. There is additional complexity and security measure introduced to ensure end-host identity, which can potentially lead to another set of security holes; While our solution is leveraging existing security measures and does not introduce new mechanisms. Further, their solution focuses on the access control policy issues rather than protecting vulnerable nodes in general. For example, routers may also be vulnerable and require protection.

There are many reasoning systems specific to firewall or NIDS. For example, filtering Postures [22] uses heuristics to automatically compute the set of filters for individual routers to enforce a particular global policy. The solution they found, however, may not be optimal. Further, they are only limited to the problem of network access control, rather than our broader goal of leveraging both filter and patch operations to mitigate network vulnerabilities. A follow-up work in [23] includes NIDS behavior into the reasoning system and differ from our work by neither considering patch operation nor trade-offs among various defense strategies.

Similar but more powerful, MulVal [24] uses formal methods to reason about the security properties which can easily enable what-if analysis such as verifying "if router A is patched, machine B will be free of attack." Our proposed framework tackles a different problem by going a step further that not only verifies that machine B is free of attack, but also computes the optimal way to stop such attack. In fact, our work complements theirs in the sense that once they finish reasoning about the vulnerabilities and identify the available options to fix the network, it can be abstracted into our model which performs the subsequent optimization.

Other works including [25, 26, 11] have somewhat similar goals though without considering patch operation either. For example, one of their goals is to find the *virtual border* - minimum number of filters or nodes to install filters. We can easily capture this goal by our *Minimal disruption* objective function. Further, we can also express other goals by using different objective functions as those listed in §4.3. The use of integer programming allows us to easily accommodate new objective functions and constraints. As a result, our framework is more general and extensible compared to previous work, as it can solve not only one particular problem but also many other problems by tuning the objective functions and taking various constraints into account.

8 Discussion

Different Types of Network-Level Defense. Different types of network defense have different capabilities (some may be able to defend against more sophisticated attacks). It is possible to distinguish different network-level defense (*e.g.,* ACL and NIDS) in our framework by assigning different cost for different types of network defense. Alternatively, we can simply always choose the most powerful defense mechanism available.

Incremental Deployment. While it is easy to use our tool to provide a new protection suggestion, our tool also fits in the scenario where the network has been partially protected and we can provide incremental suggestions in terms of additional protection based on existing setups.

Appropriate Abstraction? Note that the abstraction we have still support many of the existing abstractions. For example, to solve similar problems a human expert may use abstractions such as *the network of department x* or *the unsecured wireless network* or *the group of servers holding financial records*. We can easily support these abstractions by understanding the mapping between the group and a number of network devices or IP addresses.

Path Selection. Currently we are conservatively assuming that any path could be traversed from untrusted devices to the vulnerable device while it may not be the case in reality. One may desire to pick only paths that are in greater need of protection by ranking each path by the probability that it is selected as the actual forwarding path. This can be done by enumerating all possible failures in the network and simulate the routing algorithm to find the path [25].

9 Conclusions and Future Work

We have presented a simple and novel way of modeling the vulnerability mitigation and management problem using integer programming. We have given examples about how to model the problem. More specifically, our framework provides intelligent suggestions in terms of where to deploy filtering or where to patch which are the two main mechanisms in network defense. Further, optimal solutions can be computed by considering multiple vulnerabilities jointly which is of practical need. Our prototype suggestion tool has been evaluated using several examples based on real network topologies with demonstrated efficiency and effectiveness.

For future work, we plan to consider other objective functions and constraints. Our framework is fairly easy to extend since integer programming has a plain and clean interface. We plan to add more objective functions and constraints into our framework based on real user needs. In addition, we also plan to evaluate our tool more extensively with real usage scenarios.

References

1. Cisco IOS HTTP Server Code Injection Vulnerability, http://tools.cisco.com/security/center/viewAlert.x?alertId=10102
2. Cisco IOS Software UDP Packet Processing Denial of Service Vulnerability, http://tools.cisco.com/security/center/viewAlert.x?alertId=17765
3. Cisco Intrusion Prevention System, http://www.cisco.com/en/US/products/sw/securesw/ps2113/index.html
4. Multiple Vendor DNS Implementations Insufficient Entropy Vulnerability, http://tools.cisco.com/security/center/viewAlert.x?alertId=16183
5. Grote, A., Funke, R., Heiss, H.-U.: Performance evaluation of firewalls in gigabit-networks. In: Proc. 1999 Symposium on Performance Evaluation of Computer and Telecommunication Systems (1999), http://www.kbs.cs.tu-berlin.de/publications/fulltext/GFH99.pdf
6. Capretta, V., Stepien, B., Felty, A., Matwin, S.: Formal correctness of conflict detection for firewalls. In: FMSE 2007: Proceedings of the 2007 ACM Workshop on Formal Methods in Security Engineering, pp. 22–30 (2007)

7. Introduction to Cisco Inventory and Reporting, http://www.cisco.com/en/US/docs/net_mgmt/inventory_and_reporting/User_Guides/Introduction_to_Cisco_Inventory_and_Reporting.html
8. David System, a network management system (nms), http://www.hadden.pl/en/index.php
9. Introduction to OVAL: A new language to determine the presence of software vulnerabilities (2003), http://oval.mitre.org/documents/docs03/intro/intro.html
10. Cisco Intellishield, http://www.cisco.com/security/
11. Todtmann, B., Rathgeb, E.P.: Integrated management of distributed packet filter configurations in carrier-grade ip networks. In: International Conference on Networking, p. 44 (2007)
12. NetMRI, http://www.netcordia.com/
13. Cisco Multiple Vulnerabilities, http://secunia.com/advisories/23867/
14. Old, J.L., Buchanan, W., Graves, J., Saliou, L.: Performance analysis of network based forensic systems for in-line and out-of-line detection and logging. In: 5th European Conference on Information Warfare and Security, ECIW (2006)
15. CPLEX, High-performance software for mathematical programming and optimization, http://www.ilog.com/products/cplex/
16. GTITM, Modeling Topology of Large Internetworks, http://www.cc.gatech.edu/projects/gtitm/
17. Bartal, Y., Mayer, A., Nissim, K., Wool, A.: Firmato: A novel firewall management toolkit. ACM Trans. Comput. Syst. 22(4), 381–420 (2004)
18. Mayer, A., Wool, A., Ziskind, E.: Fang: A firewall analysis engine. In: SP 2000: Proceedings of the 2000 IEEE Symposium on Security and Privacy, p. 177 (2000)
19. Al-shaer, E., Hamed, H., Boutaba, R., Hasan, M.: Conflict classification and analysis of distributed firewall policies. IEEE Journal on Selected Areas in Communications 23, 2069–2084 (2005)
20. Bellovin, S.M.: Distributed firewalls. Login, 37–39 (1999)
21. Ioannidis, S., Keromytis, A.D., Bellovin, S.M., Smith, J.M.: Implementing a distributed firewall. In: CCS 2000: Proceedings of the 7th ACM Conference on Computer and Communications Security, pp. 190–199 (2000)
22. Guttman, J.D.: Filtering postures: local enforcement for global policies. In: SP 1997: Proceedings of the 1997 IEEE Symposium on Security and Privacy, p. 120. IEEE Computer Society (1997)
23. Uribe, T.E., Cheung, S.: Automatic analysis of firewall and network intrusion detection system configurations. In: FMSE 2004: Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering, pp. 66–74 (2004)
24. Ou, X., Govindavajhala, S., Appel, A.W.: Mulval: a logic-based network security analyzer. In: SSYM 2005: Proceedings of the 14th Conference on USENIX Security Symposium (2005)
25. Tödtmann, B., Rathgeb, E.P.: Anticipatory distributed packet filter configurations for carrier-grade ip networks. Comput. Netw. 51(10), 2565–2579 (2007)
26. Todtmann, B., Rathgeb, E.P.: Advanced packet filter placement strategies for carrier-grade ip-networks. In: AINAW 2007: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops, vol. 1, pp. 415–423 (2007)