Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Tesi di Laurea Specialistica

Complex Attack Analysis and Safeguard Selection: a Cost-Oriented Approach

Suk Wah Cristina Tang

Relatore

Prof. Fabrizio Baiardi

Anno Accademico 2007/2008

Abstract

When intelligent threats attack a system, they rarely achieve their goals by exploiting a single vulnerability. Rather, they achieve their goals by composing attacks and by exploiting structural security flaws of the target system. Attack graphs have been the de facto tool for discovering possible complex attacks. This thesis proposes a cost-effective safeguard selection strategy, which first identifies a complex attack set that covers all the complex attacks through the use of attack graphs and later selects a minimal set of countermeasures through the formulation and resolution of an integer linear programming problem. Multiple goals in conjunction or disjunction relation can be analyzed. We have built a working prototype system that implements this strategy and that helps maximizing the return-on-investment by identifying critical stepping-stone hosts and by suggesting the most cost-effective set of countermeasures. The mechanism of this approach is independent of the modeling abstraction level. We have considered both an example model that goes into the details of elementary attacks and an example model that targets worst-case analysis.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Professor Baiardi, for his insightful advices and patient guidance in the preparation of this thesis.

I am in debted to my friends, my family and my family-in-law, at home and abroad, for all their love and care during these years.

Finally, I wish to present my dearest thanks to Massimo. I could have never accomplished this thesis without his encouragement and unselfish love.

Contents

Chapter 1

Introduction

As the Internet becomes a tangle of virtually all computer networks, defending an enterprise network is more and more challenging. The advance in technology only leverages the task of intruders because attacks that were once impossible become not only possible, but even automatable. As prebuilt exploits are readily available, an intruder often only has to assemble existing exploits to execute a complex attack that was accessible exclusively to hard-core hackers. The diffusion of information helps intruders to learn about new vulnerabilities and attacks. The game of defending a network is unfair to begin with because the intruder only needs to know *one* complex attack while the defender needs to defend from *all* complex attacks.

Another difficulty of defending a network arises because of organization vulnerabilities. Most users of computers in a network are unaware of almost all security risks and they run vulnerable clients [**6**] that are great entry points to an enterprise network. As soon as an intruder compromises and takes control over some clients, she can then use them as stepping stones [**25**] and take advantage of client/server trust relationships to execute other attacks that would be otherwise impossible. A typical example is a client that has a certain privilege on a database on an internal server. In this case, after overtaking the client, the intruder can use the client privileges to compromise the confidentiality and integrity of the database. In this scenario, traditional vulnerability scanners like SATAN [**11**] and Nessus [**4**] are no more sufficient to discover complex attacks that are enabled by combining apparently harmless and

unrelated vulnerabilities.

Firewalls are still among the main tools in the defensive arsenal of system administrators. However, firewalls alone can rarely achieve sufficient protection because of the large number of protocols and encrypted traffic. As most complex attacks exploit structural security weakness, tools that analyze the interrelationships among elementary attacks are invaluable. Among these tools, attack graphs are well known. An attack graph shows all the sequences of elementary attacks that an intruder can implement to achieve a set of goals because each edge corresponds to an elementary attack. Each graph node represents a snapshot of the security state of the infrastructure [**18**] and a complex attack is a path in the graph. By analyzing the states crossed by the path that corresponds to a complex attack, it is possible to understand exactly *how* the attack happens. Attack trees are closely related diagrams, where each node represents an attack, but they are more focused on the *composition* of attacks because each internal node shows how an attack can be decomposed. There are two types of internal nodes in an attack tree: AND nodes and OR nodes. An AND node n requires the execution of all the attacks represented by its children nodes in order to execute the attack represented by n. An OR node n' requires the execution of at least one of the attacks represented by its children in order to execute the attack represented by n'. Attack trees show *why* a complex attack happens because the tree structure reveals the causality between attacks. The approach adopted in this thesis focuses in answering the question "*how to stop* all the complex attacks" where countermeasures play a central role. We will see in later sections that under the monotonicity assumption [**8, 13, 12**], which assumes that an attacker never loses any acquired right, it is possible to answer this question without knowing *how* and *why* a complex attack happens. The question is answered in two steps: the first one computes a complex attack set, which covers all the complex attacks. The second one computes a minimal set of countermeasures that stops all the complex attacks in the previous set.

While there are numerous proposals of attack graph generators [**18, 22, 10, 19, 8, 12, 20, 23**], here we propose yet another attack graph analyzer. The most significant difference with respect to other tools is the emphasis on safeguard selection and the separation of the domain specific problem and the underlying problem. The tractability of the analysis of an enterprise network depends upon careful modeling

of the network, which requires expertise in that area. The network and, in consequence, the elementary attacks, should be modeled with just the necessary details to create a sufficiently expressive model without hindering the tractability of the analysis. As a side effect of separating the model from the underlying problem, it is possible to create model transformers that automatically generate complementary views of the same network at different abstraction levels. Our focus is on the underlying problem: given a set of initial capabilities and a model described as a set of rules, find a minimal set of countermeasures that stops all the complex attacks. In terms of attack graphs, this can be expressed as the computation of a minimal cut set that includes at least one edge for each complex attack. If all the attacks corresponding to the edges in the cut set are stopped, then no complex attack is feasible.

The thesis has developed a working prototype system that has been tested on both small case studies and large simulated networks. Experimental results on simulated networks show that this prototype system can handle networks with thousands of hosts. However, the results also show that modeling the same network at distinct detail levels can have drastic impact on the efficiency of the analysis. This enforces our belief that separating the model is vital for the feasibility of analysis on real complex networks where the modeling is better left to domain experts.

This thesis is organized as follows:

Chapter 2 - Attack graph

This chapter introduces the formal definitions of terms that will be used throughout this thesis.

Chapter 3 - Attack Graph: State of the Art

This chapter describes alternative approaches to attack graph analysis that have been proposed in the literature.

Chapter 4 - Algorithms

This chapter describes the algorithms that have been developed in this thesis, their complexity and correctness. It also describes how the problem of finding a minimal cut set can be modeled as an integer linear programming problem. Two simple case studies show some examples of network modeling and exemplify the functioning of the algorithms.

Chapter 5 - Attack Graph Toolkit

This chapter describes the specification and implementation of the toolkit.

Chapter 6 - Evaluation

This chapter describes the simulated networks and the experimental results. An example of different views of the same network illustrates the impact of modeling at different abstraction levels on efficiency.

Chapter 2

Attack Graph

An attack graph describes a set of complex attacks against an infrastructure where each complex attack is composed of several elementary attacks. There are alternative representations of attack graphs and the efficiency of the generation of these graphs depends upon the chosen representation. However, these alternative representations contain essentially the same information, namely all the steps that an attacker has to implement in order to achieve a predefined set of goals. This chapter defines the terminology used throughout this thesis and introduces our chosen representation.

2.1. Definitions

Definition 2.1.1. Information Infrastructure

An information infrastructure consists of a set of interconnected entities. An entity represents one of several components, among them a physical host, a virtual host, a set of hosts, a router, a firewall. In other words, an entity is an end-point of a logical connection, typically at the IP level, or of a physical one. Entities are connected through interfaces. Each interface of an entity is connected to at least one interface of another entity and each entity has at least one interface. □

In order to define the infrastructure model, we have to define entities. It is important to choose the proper level of abstraction, as the complexity of generating an attack graph is greatly influenced by the number of entities. In general, the proper level of abstraction depends upon the context. Suppose, as an example that we are interested in performing a what-if analysis to evaluate alternative firewall configurations and the adoption of an IDS. In this case, the most appropriate choice may be the one that defines an entity as a set of hosts of a subnet.

Definition 2.1.2. Conditions

A condition is a predicate on the state of the information infrastructure at a certain instant. C, the set of all conditions of interest is finite and is ranged over $c_1, c_2, c_3, ..., c_m$. □

A condition can be a predicate on:

- ❖ a property of an entity
- ❖ a relationship among entities

The correctness of the approach described in subsequent sections depends upon the monotonicity assumption, which requires that:

(1) after a condition has become true it cannot be reverted to false
(2) no condition is the negation of another one

The monotonicity assumption [**8, 13, 12**] simplifies the search of complex attacks. There are three reasons to support the adoption of this assumption:

- ❖ most attacks can be modeled as monotonic with reasonable fidelity [**8**];
- ❖ with respect to a non-monotonic model, a monotonic one corresponds to the worst case where effects of attacks are non reversible. Hence, analyses that consider this model may yield false positives but not false negatives, i.e., it may find complex attacks that are actually infeasible but it never miss any effective attack;

- ❖ the monotonic model greatly simplifies the problem and, therefore, it enables the analysis of larger networks.

The conditions in C are concrete and are not parametric. They are instantiated from templates such as "privilege x on entity y".

Example 2.1.3. Example conditions regarding a property of an entity

- ❖ root-level privileges on entity A
- ❖ control over the confidentiality of entity A
- ❖ the existence of a vulnerability on entity A □

Example 2.1.4. Example conditions regarding a relationship among entities

- ❖ entity A can access to port 21 of entity B
- ❖ entity A trusts entity B, that is, users of entity B can connect to entity A without authentication □

Definition 2.1.5. State

The state of the information infrastructure at a certain instant is characterized by the conditions that hold at that instant. □

Definition 2.1.6. Rule

A rule is a pair $< preconditions, postconditions >$ where

- ❖ *preconditions* is a set of conditions that must be true for the rule to be applicable;

- ❖ *postconditions* is a set of conditions that will be true after the application of the rule. If a postcondition is true before the application of a rule then it will remain true.

Let R be the finite set of rules and be ranged over $r_1, r_2, r_3, ..., r_n$. We assume that R is known. A rule can be applied in a state if the rule preconditions are true in the state. □

Definition 2.1.7. Dependency

A dependency is a pair $< C_1, C_2 >$ where

- ❖ C_1 and C_2 are sets of conditions
- ❖ the validity of all the conditions in C_1 implies the validity of all the conditions in C_2 □

While rules that describe elementary attacks can be generated automatically, those that describe dependencies have to be defined according to the infrastructure of interest as they depend upon the interconnections among components, dictated by the infrastructure design.

Definition 2.1.8. Elementary Attack

The execution of an elementary attack corresponds to the application of a rule that describes an exploit. □

However, the application of a rule does not *necessarily* imply the execution of a exploit, as a rule may describe a dependency as well. The next example shows an application of a rule that does not correspond to the execution of an exploit.

Example 2.1.9. Consider a case where

- ❖ c_A is the condition that expresses user-level privileges on host A

- ❖ c_B is the condition that expresses user-level privileges on host B
- ❖ the rule $< \{c_A\}, \{c_B\} >$ expresses the trust relationship between A and B that says "whoever has user-level privileges on host A has user-level privileges on host B without authentication"
- ❖ the rule $< \{c_1\}, \{c_A\} >$ describes an exploit
- ❖ the initial state is $\{c_1\}$

Therefore, the rule $< \{c_1\}, \{c_A\} >$ is applicable in the initial state as the rule precondition is satisfied in that state. After its application, the postcondition c_A becomes valid and the state becomes $\{c_1, c_A\}$. This state transition models an elementary attack. Now the rule $< \{c_A\}, \{c_B\} >$ becomes applicable as its precondition is satisfied and the state becomes $\{c_1, c_A, c_B\}$. Obviously, this transition does not correspond to any elementary attack and it depends upon the relationships among the components. □

Definition 2.1.10. Goal

A goal condition is a condition that an intruder aims to achieve. A set of goal conditions are all the conditions that an intruder aims to achieve. A goal state is a state in which all the goal conditions are true. □

When there is no ambiguity, the goal refers to the set of goal conditions.

Definition 2.1.11. Initial State

The initial state is the state of the information infrastructure before the application of any rule. □

Definition 2.1.12. Final State

A final state is a state in which all the goal conditions are true. The terms final state and goal state are interchangeable. □

There can be more than one final state.

Definition 2.1.13. Attack Graph

An attack graph is a quadruple $< S, s_0, \tau, S_f >$ where

- S is a set of states
- s_0 is the initial state
- τ is a set of transitions
- S_f is a set of final states

An attack graph is always relative to a threat, which is characterized in terms of the initial conditions and the goal conditions. The set of final states includes all the final states. A state transition corresponds to the application of a rule. The graph shows all the attack sequences to achieve the goal of interest. □

The vertices of the attack graph correspond to states and the edges correspond to state transitions. A constructive definition of the attack graph can be the following:

(1) given a set of vertices V, initialized with a single vertex that represents the initial state
(2) for each vertex v in V and for each applicable rule r, add an outgoing edge labeled r from v to v', where v' is the state after the application of r. If v' is not already present in V, a new vertex will be created
(3) repeat the above step until no more new edge is found

While this process illustrates the conceptual strategy, we recall that such a construction is inefficient and does not take place in practice.

Definition 2.1.14. Complex Attack
A complex attack is a path in the attack graph from the initial state to a final one. □

Some practical examples of elementary attacks are:

- ❖ buffer/stack overflow
- ❖ sniffing
- ❖ connection stealing
- ❖ replay attack

A rule can either be an elementary attack or a dependency, this differs from the approach that uses a separate dependency graph [**10**]. In the first case, rules that describe dependencies may be part of a complex attack. In the latter case, complex attacks are defined exclusively in terms of elementary attacks. The advantage of modeling dependencies as rules is that countermeasures, which define how attacks can be stopped, may refer to dependencies. For example, if a complex attack requires the exploitation of a vulnerability on host h_1 and the abuse of the dependency between host h_1 and host h_2, then a countermeasure for this complex attack may remove the dependency. On the contrary, the separation of dependencies in a dependency graph implies that dependencies are non-modifiable and therefore, should not be considered during the safeguard selection stage.

The following example shows that the two modelings, namely modeling dependencies as rules and modeling dependencies through the dependency graph, return different results that describe the same complex attack.

Example 2.1.15. Consider two rules $r_1 \equiv < \{c_1\}, \{c_{I_k}\} >$ and $r_2 \equiv < \{c_{I_k}\}, \{c_{c_k}\} >$ that represent respectively

- ❖ an elementary attack that has a single precondition c_1 and a single post-condition c_{I_k} , which is control over the integrity of the component k
- ❖ a dependency that says, c_{c_k} , the confidentiality of the component k, depends upon c_{I_k} , the integrity of the component k

Suppose that c_1 is one of the initial conditions and that c_{c_k} is the goal. If the dependency is modeled as a rule, then a complex attack is the composition of r_1 and r_2, and only after the application of the latter a goal state is reached. If the dependency graph is separated from the attack graph, then the complex attack is composed of only r_1 and the state after the application of r_1 will be a goal state because of the transitive closure [**10**] on the dependency graph. □

We will refer to complex attacks simply as attacks when there is no risk of ambiguity. Under the assumption of monotonicity, we only need to consider acyclic paths. In general, an attacker, i.e., a threat, composes elementary attacks to exploit trust relationships between hosts, so as to achieve privileges that cannot be gained through attacks on a single host.

The following example shows the case of a cyclic path.

EXAMPLE 2.1.16. The path shown in this example is cyclic as the node representing the state s_2 is passed through twice.

$s_0 \xrightarrow{r_1} s_1 \xrightarrow{r_2} s_2 \xrightarrow{r_4} s_2 \xrightarrow{r_3} s_g$ where

- ❖ s_0 is the initial state, e.g., $\{c_1\}$
- ❖ s_g is a final state if c_3 is the goal condition
- ❖ $s_1 = \{c_1, c_4\}$
- ❖ $r_1 =< \{c_1\}, \{c_4\} >$
- ❖ $r_2 =< \{c_4\}, \{c_2\} >$
- ❖ $r_3 =< \{c_2\}, \{c_3\} >$
- ❖ $r_4 =< \{c_4\}, \{c_1\} >$

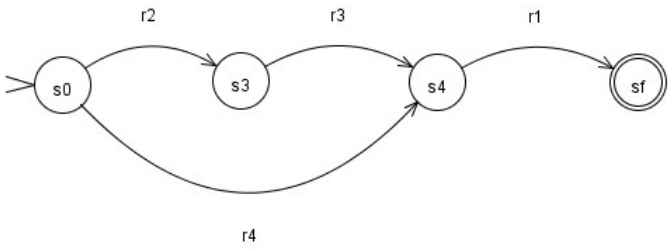
Two states are identical if they include exactly the same set of conditions. We observe that under the monotonicity assumption, all the states in a cycle are identical. □

Definition 2.1.17. Countermeasure

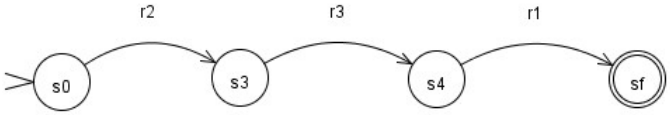
A countermeasure for an elementary attack inactivates a rule, i.e., it prevents the application of the rule. A countermeasure for a complex attack inactivates at least one rule that belongs to the complex attack so that the path corresponding to a complex attack is split into two segments, where the initial state and the final state belong to different segments. Therefore, the path is no more viable. □

We assume that we can introduce a countermeasure for each elementary attack. Our aim is to find a set of countermeasures so that no final state can be reached. Distinct complex attacks may share the same countermeasure as shown in Ex. 2.1.18.

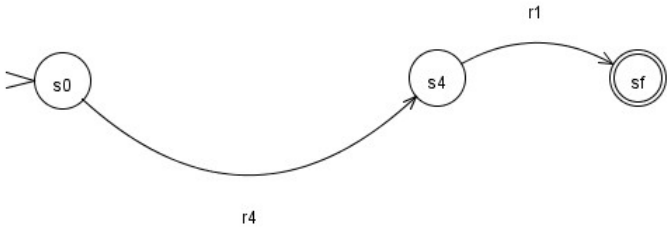
Example 2.1.18. The following attack graph fragment describes two complex attacks $r_2r_3r_1$ and r_4r_1:



The first complex attack is $r_2r_3r_1$.



The second complex attack is r_4r_1.



Suppose that $c_{r_2}, c_{r_3}, c_{r_4}, c_{r_1}$ are the countermeasures for the rules r_2, r_3, r_4, r_1, respectively. Alternative countermeasures for the attack $r_2 r_3 r_1$ are c_{r_2}, c_{r_3} and c_{r_1}, while those for the attack $r_4 r_1$ are c_{r_4} and c_{r_1}. In this example, if we choose c_{r_1} as the countermeasure for both attacks, then they will share the same countermeasure. □

Some examples of countermeasures are:

- ❖ installing a new version of a program
- ❖ applying a patch
- ❖ eliminating a service relative to a vulnerability
- ❖ modifying dependencies between components, i.e., between hosts

The result returned by the toolkit described in this thesis only suggests *which* rules to be inactivated. Since, in general, there are alternative ways to inactivate a rule, the choice of the strategy to be adopted to inactivate a rule is left to users of the toolkit. For example, there are multiple ways to eliminate a stack overflow vulnerability:

- ❖ the adoption of strong typing systems
- ❖ checking the length of input strings
- ❖ making the stack non-executable

Definition 2.1.19. Complete and minimal set of countermeasures

A set of countermeasures Σ is *complete* if any complex attack is no more feasible after the countermeasures in Σ have been adopted. A set of countermeasures Σ is *minimal* if none of its proper subset is complete [**10**]. □

Properties of interest of attack graphs:

- ❖ an attack graph is *exhaustive* [**19**] if it describes any possible complex attack
- ❖ an attack graph is *succinct* [**19**] if it describes *only* those states from which a final state can be reached

The attack graph constructed as described in paragraph 2.1 is exhaustive but not succinct. The approach of Sheyner *et al.* [**22**] applies the model checker NuSMV [**5**] to build a succinct attack graph by removing the vertices that do not lead to a final state. The main problem of this approach is scalability. Lippmann and Ingols [**16**] have shown that the tools described in the paper have been used on infrastructures with at most 20 components and that they do not scale adequately in the case of infrastructures with hundreds of hosts. All these tools share a common characteristic: the generation of all the attack *paths*. If we observe the usage of attack graphs, we will notice that their main use is to help system administrators to select countermeasures to be adopted [**18**]. Bearing this in mind, the approach we advocate exploits the fact that we are interested in finding all the elementary attacks used to build complex attacks, but this does not implies that we are interested in the *sequences* to compose elementary attacks. Therefore, the proposed toolkit generates a complex attack set Ω bypassing the generation of an attack graph.

Definition 2.1.20. Complex Attack Set Ω

A complex attack set Ω is a set $\{S_1, ..., S_n\}$ where each S_i, $i = 1..n$, is a set of rules and it corresponds to some complex attacks. It is complete if all the complex attacks correspond to some elements of Ω. □

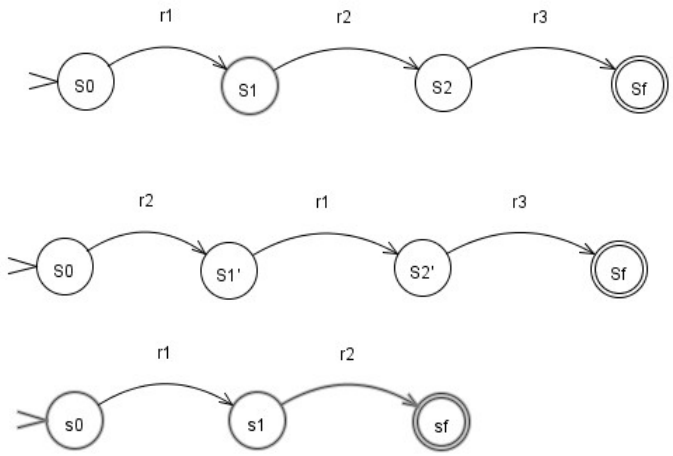
Each complex attack is a path in the attack graph, a path consists of a sequence of edges and each edge corresponds to a rule. Therefore, a complex attack α can be represented as a set of rules R_α, which includes the rules that belong to the path. A complex attack α is *covered* by a set X whose elements are set of rules if there is an element x of X such that:

- ❖ x is non-empty
- ❖ x is a subset of R_α

Formally, α is covered by X if $\exists\ x \in X$ such that $x \neq \emptyset \wedge x \subseteq R_\alpha$. We are interested in finding a complex attack set Ω that is *complete*, that is, it covers every complex attack.

The same set of rules can cover distinct complex attacks, as shown in Ex. 2.1.21.

Example 2.1.21. The following three complex attacks are covered by the same element $\{r_1, r_2\}$ of the set $\{\{r_1, r_2\}\}$.



The first and second complex attacks are both composed of r_1, r_2, r_3 and $\{r_1, r_2\}$ is a non-empty subset of $\{r_1, r_2, r_3\}$. The third complex attack is composed of r_1, r_2 and $\{r_1, r_2\}$ is a non-empty subset of $\{r_1, r_2\}$. Therefore, all these three complex attacks are covered by $\{\{r_1, r_2\}\}$. □

Chapter 3

Attack Graph: State of the Art

This chapter reviews works on the topic of complex attack analysis. The goal of this analysis is to give an overview of all the strategies available to an attacker to achieve a predefined goal [**22**]. Attack graphs [**16**] are recognized as a valuable tool to discover these complex attacks. All the terms used in this section have been formally defined in Chapt. 2.

Attack graphs are intrinsically complex because of the state explosion problem and several papers [**8, 13, 18, 22, 19**] proposed alternative solutions to tackle this problem. This section investigates some of these ideas in details. For reference, Lippmann *et al.* [**16**] have presented a review of current attack graph related tools.

It is well accepted that the manual construction of attack graphs is impractical even for a moderately sized graph [**22**] because of the ever changing nature of information infrastructures and the effort required to reconstruct the graphs. There are various tools that generate attack graphs automatically, most of these tools assumes that the input data for the construction is known [**10, 8, 24, 19**].

This data includes:

- ❖ the list of vulnerabilities of all the components
- ❖ the dependencies among components
- ❖ the list of known attacks

While these data may not be immediately available in a proper format, the implementation of NetSPA [**12**] has shown that this information can be readily imported from common sources such as CVE dictionary [**1**], NVD [**15**] and output of Nessus scans [**4**], supporting that our assumption is grounded.

3.1. Approach based on Attack Graph Simplification

As the complete attack graph is often too large to be visualized, there are various attempts [**18, 12, 7**] to simplify it. Ingols *et al.* [**12**] proposed the multiple-prerequisite graph, which differs from the usual attack graph in a number of ways:

- ❖ the multiple-prerequisite graph has a non-tree structure[1], which contains cycles and nodes with multiple incoming edges. As a consequence, a common subpath may be shared among distinct paths in the representation. In the example shown in Fig. 3.1.2, the subpath a_3, a_4 is shared instead of being represented as part of two distinct paths a_1, a_3, a_4 and a_2, a_3, a_4 as in Fig. 3.1.1 .

- ❖ the multiple-prerequisite graph merges nodes of the same class. A class is defined as follows:
  - ◇ each node n belongs to a class
  - ◇ two nodes n_1, n_2 belongs to the same class if and only if
    - * $C_1 = \{c \mid$ c is the class of n_{prec}, n_{prec} is a predecessor of $n_1\}$
    - * $C_2 = \{c \mid$ c is the class of n_{prec}, n_{prec} is a predecessor of $n_2\}$
    - * $C_1 = C_2$
    - * $C'_1 = \{c \mid$ c is the class of n_{succ}, n_{succ} is a successor of $n_1\}$
    - * $C'_2 = \{c \mid$ c is the class of n_{succ}, n_{succ} is a successor of $n_2\}$
    - * $C'_1 = C'_2$

- ❖ a node of the multiple prerequisite graph may represent

[1] For simplicity we name the edges here even if in a multiple-prerequisite graph they are not named.

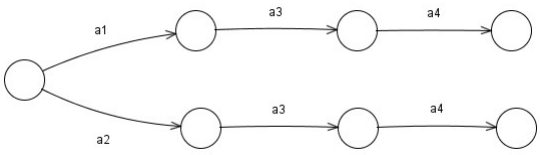


FIGURE 3.1.1. The paths a_1, a_3, a_4 and a_2, a_3, a_4 are distinct.

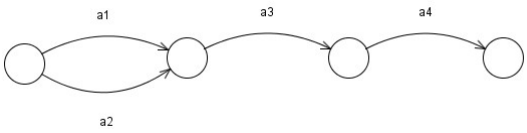


FIGURE 3.1.2. The subpath a_3, a_4 is shared.

- ◇ a state where a state is represented as $\{< host, access\ level >\}$ and the access level can be
  - ∗ root: host is reachable (root-level privileges)
  - ∗ user: host is reachable (user-level privileges)
  - ∗ DoS: denial-of-service
  - ∗ other: confidentially and/or integrity loss
- ◇ a vulnerability instance: while attack graphs model a vulnerability instance as an edge, the multiple prerequisite graph models it as a node. As an example, the multiple prerequisite graph represents the edge $s_0 \xrightarrow{v_1} s_1$ in an attack graph as $s_0 \rightarrow v_1 \rightarrow s_1$.
- ◇ a set of prerequisites: the preconditions for an attack to take place. An attack graph does not explicitly model the prerequisites, which are conditions to be checked to determine whether an attack can be executed. The multiple prerequisite graph explicitly represents a precondition as a node. Hence, a path is always built by composing segments with the structure:
  state node → *prerequisite node* → *vulnerability instance node*

Several node types are introduced to simplify the generation of reachable nodes. For example, if only one type of nodes, state nodes, is introduced, then in order to compute the reachable nodes from a node s_i, we have to find:

- ❖ all the vulnerabilities present in s_i
- ❖ all the attacks enabled by the previous vulnerabilities
- ❖ all the states reachable by the execution of the enabled attacks

By defining several node types, the results of these intermediate operations can be *precalculated* and stored in one or several arrays. Hence, the set of reachable states from a state s_i can be computed by retrieving the proper values from the precalculated arrays as follows:

- ❖ $c = S2C(s_1)$ where S2C is an array that maps states into credentials
- ❖ $v = C2V(c)$ where C2V is an array that maps credentials into vulnerability instances enabled by those credentials
- ❖ $reachable\,states = V2S(v)$ where V2S is an array that maps vulnerability instances into states

If Q is a queue of nodes that is initialized with the root node, the following procedure builds the multiple prerequisite graph:

- ❖ while Q is not empty
  - ◇ remove a node n from Q
  - ◇ for each node r that can be reached from n
    - ∗ add an edge from n to r
    - ∗ if r does not belong to Q, then add r to Q

Under the assumption that the arrays are precalculated and that are accessible in constant time, the reachable set of nodes can be calculated in constant time as well and, therefore, the complexity is linear in the number of nodes.

3.2. Approach based on Model Checking

Model checking is a method to verify properties of a formal system. Given a model M and a property P, the model checker verifies that M satisfies P. In general, if P is not satisfied, the model checker outputs a counterexample to show how P can be

violated. In the context of attack graphs, the model describes the evolution of an information infrastructure, the property of interest is a safety property regarding of state of the infrastructure. If the safety property is violated, the counterexample represents a successful attack. The main problem of this approach is that a model checker does not tell us *all* the possible attacks that result in a violation of a safety property.

To overcome this disadvantage, Sheyner *et al.* [**22**] have exploited another functionality of model checkers: the one that computes all the states from which it is possible to reach a state where a property P is satisfied and where P is defined as the negation of the safety property SP of interest. For example, if SP is "no root-level privileges" then P is defined as the negation of SP, i.e., "root-level privileges". Suppose that s_p is a state where P is satisfied, then the model checker outputs a set of states $S = \{s' | \, s_p \; is \, reachable \, from \; s'\}$. Hence, the approach of Sheyner *et al.* [**22**] applies the model checker in parallel to the construction of the attack graph G to find the set S. When these operations have been completed, S is used to prune G. In this way, the pruned attack graph contains only those nodes and edges that are related to some complex attacks. As noted by the authors, this approach suffers from scalability problem because of the performance bottleneck due to the generation of the attack graph G. According to the tests described in the paper, the time to generate G is orders of magnitudes larger than the one to generate S. Another test of this tool, described in Ou *et al.* [**18**], shows that in the case of an information infrastructure including 10 hosts, with 5 vulnerabilities each, the tool requires 15 minutes to generate an attack graph with 10 millions edges. Another observation is that the attack graph G is computed independently of the generation of the set S by the model checker, even if the two computations share several sub computations.

3.3. Other approaches

Since the complexity of an attack graph is mainly influenced by the number of components, Baiardi *et al.* [**10**] proposed a hierarchical approach. In their approach, at first the information infrastructure is defined as a coarse grain model, where the

components are abstract entities. Subsequently, each component is considered as an infrastructure so that the information infrastructure may be repeatedly decomposed in this manner. Conceptually, the information infrastructure is observed through a magnifying lens of increasing power.

The dependencies among components are modeled as directed hyperedges in a hypergraph, the *Infrastructure Hypergraph*. An hyperedge has one head and one or several tails, each head/tail is labeled with one of the three security attributes: a, c, i which stand respectively for *availability*, *confidentiality* and *integrity* of the component connected to that head/tail. For example, $C_1 \, ^a \rightarrow^i C_2$ means that the integrity of the component C_2 depends upon the availability of the component C_1.

Under certain conditions, the components of the decomposed infrastructure can be analyzed in parallel. Therefore, each component can be considered as a black box during the infrastructure analysis. When a new component is introduced, the results of the analysis of other components can be reused. Hence, the analysis needs to be applied only to the new components and to the edges that relate the new component to the old ones. When a component C_1 is decomposed into subcomponents $C_1^1, C_1^2, ...C_1^n$, the old edges of the Infrastructure Hypergraph relative to C_1 are replaced by edges that link directly to the subcomponents.

The evolution of the information infrastructure is described by the Evolution Graph, which is orthogonal to the Infrastructure Hypergraph. Each node of the Evolution Graph represents a set of rights
where

- a right is a pair $< component, attribute >$
- an attribute can be c, a, i which stand for confidentiality, availability and integrity, respectively

The transitions of states of the Evolution Graph are triggered by successful executions of elementary attacks. An attack A is a triple
$< component, preconditions(A), postconditions(A) >$ where both $preconditions(A)$ and $postconditions(A)$ are sets of rights.

EXAMPLE 3.3.1. Suppose that

- ❖ the state is $\{< C_1, a >, ...\}$ at a certain instant
- ❖ there is an attack $A \equiv < C_1, \{< C_1, a >\}, \{< C_2, a >\} >$

After the execution of A, the state becomes $\{< C_1, a >, < C_2, a >, ...\}$. □

This example shows that in order to execute an attack A in a state s, the preconditions of A have to be satisfied in s, i.e., $s \supseteq preconditions(A)$. After the execution of an attack, the rights the attacker has gained will be added to those in the previous state to define the new state s', i.e., $s' \supseteq postconditions(A)$. In this way, the rights increase monotonically. After each transition, the state is updated by computing the transitive closure of rights using the information from the Infrastructure Hypergraph.

EXAMPLE 3.3.2. Suppose that the Infrastructure Hypergraph contains a dependency $C_2 \, ^a\!\!\rightarrow^c C_3$, then the execution of the attack A will update the state to $\{< C_1, a > , < C_2, a >, < C_3, c >, ...\}$. Suppose now that

- ❖ the component C_2 is decomposed into C_2' and C_2"
- ❖ the goal is $< C_3, c >$

Two cases are possible:

(1) the attack A and the dependency both refer to the same subcomponent of C_2, without loss of generality let C_2' be that subcomponent. Then, the attack A can still be executed.

(2) the attack A refers to the subcomponent C_2' whereas the dependency refers to C_2". There are two cases:

   (a) there exists some dependencies inside C_2 so that $< C_2", a >$ belongs to the transitive closure of $< C_2', a >$. Therefore, the attack can still be executed;

   (b) $< C_2", a >$ does not belong to the transitive closure of $< C_2', a >$. In this case, the attack cannot take place and hence, the two views of the same Infrastructure Hypergraph at two distinct detail levels are inconsistent. □

The pitfall of this approach is that the analysis that considers the Infrastructure Hypergraph at an abstract level may return false positives, that is, it may return complex attacks that are actually infeasible. The main idea of this approach is to reuse partial results of the analyses to minimize the overall complexity.

Chapter 4

Algorithms

This chapter introduces the strategy to generate a minimal set of countermeasures whose application makes any complex attack ineffective. This strategy consists of two steps. The first step generates a complex attack set Ω which covers all the complex attacks. This set is computed in a two-phase algorithm bypassing the generation of the whole attack graph. The two phases are:

- Phase I: it finds all the reachable conditions given the initial conditions and it is based upon forward reasoning
- Phase II: it generates the complex attack set and it is based upon backward reasoning

The main advantage of introducing a two-phase approach is the possibility to reuse the information generated in Phase I to analyze distinct goals of the attacker because the results of Phase I is independent of the goals. Furthermore, multiple instances of Phase II can be executed in parallel. As exemplified in Ingols *et al.* [**12**], in the case of a network with 252 hosts, even a simplified attack graph, which is shrank by 99% with respect to the full attack graph, is still too complex to be interpreted. Therefore, we do not attempt to generate the attack graph which includes far more information than necessary. Instead, we focus on generating the extractable information from the attack graph that is useful to the safeguard selection process. The complex attack set captures this information.

The second step computes a minimal set of countermeasures that stops all the complex attacks. The problem of selecting a set of rules that includes at least one rule from each element of the complex attack set is modeled as an integer linear programming problem. The minimal set of countermeasures includes countermeasures corresponding to the selected rules. This set will be further reduced by selecting the most favorable countermeasures if there are some budgetary constraints. As the reduced set of countermeasures is no longer complete, complementary countermeasures through modifications to security policies are recommended to get rid of the remaining security holes.

4.1. Phase I: Forward Computation of the Set of Reachable Conditions

Our aim is to find the set of all the reachable conditions so that only viable paths are considered in Phase II. A condition c is reachable if there exist some rules through whose application c becomes true.

4.1.1. Inputs

- ❖ the current set of reachable conditions $\mathbb{C}$, which is initialized to the set of initial conditions C_0
- ❖ a set of rules R

We assume that any rule with n postconditions ($n > 1$) has already been split into n rules. All these rules share the same identification so that the generated countermeasures refer to the original rule. As an example, the rule $< \{c_1, c_4\}, \{c_2, c_3, c_5\} >$ with ID 18 will be split as $< \{c_1, c_4\}, \{c_2\} >$, $< \{c_1, c_4\}, \{c_3\} >$ and $< \{c_1, c_4\}, \{c_5\} >$ and all of them share the same ID 18.

Algorithm 4.1.1. *Reachable Conditions Generation (R, C_0)*

The algorithm repeats the following steps until either R is empty or no more rule is applicable

(1) mark all the rules $R' \subseteq R$ which are applicable with the current set of reachable conditions $\mathbb{C}$
(2) add the postconditions of the marked rules R' to $\mathbb{C}$
(3) remove R' from R □

A high level description of this algorithm is shown in Fig. 4.1.1.

4.1.2. Complexity

The worse case time complexity of the algorithm is $O(n^2)$ where n is the total number of rules. This case occurs if all the rules form a chain where the postcondition of the i^{th} rule is the precondition of the $(i+1)^{th}$ one, for $i = 1..n-1$. In practice, we expect a much lower average complexity as C grows monotonically and the number of applicable rules increases with the size of C. The space complexity is $O(n)$.

The rule splitting described in Sect. 4.1 does not increase the complexity. With reference to the previous example, if $< \{c_1, c_4\}, \{c_2, c_3, c_5\} >$ is applicable, then $< \{c_1, c_4\}, \{c_2\} >$, $< \{c_1, c_4\}, \{c_3\} >$ and $< \{c_1, c_4\}, \{c_5\} >$ will be applicable as well. Hence, each of the decomposed rules will be visited the same number of times as the original rule.

4.2. Phase II: Backward Generation of the Complex Attack Set

In this phase, the algorithm computes the complex attack set by searching backwards from the goal condition. Before describing the algorithm, we have to introduce some definitions.

Definition 4.2.1. Minimal subset set

A minimal subset set S is a set of sets $\{s_1, s_2, s_3, ...\}$ where no element of S is a proper subset of another element of S. Formally, $\forall s_i, s_j \in S.\, s_i \subseteq s_j \leftrightarrow s_i = s_j$. □

```
1  // R: the set of rules, each rule has two fields:
2  //     preconditions of type {conditions}
3  //     postconditions of type {conditions}
4  // C: all the current valid conditions,
5  //     initially C = {initial conditions}
6
7  reachableConditions (R, C)
8    modified := true
9    while (R != Empty && modified = true) {
10     modified := false
11     newConditions := EmptySet
12     // we want to update C only after scanning all the rules in R
13     foreach r in R {
14       if (r.preconditions in C) {
15         modified := true
16         R.remove(r)
17         newConditions.add(r.postcondition)
18       }
19     }
20     C := C `union` newConditions
21   }
22   return C
```

FIGURE 4.1.1. high level algorithm to compute reachable conditions

$\perp$ is a special minimal subset set. For any minimal subset set S, the union of $\perp$ and S is $\perp$.

EXAMPLE 4.2.2. $\{\{1,2\},\{2,3\}\}$ is a minimal subset set while $\{\{1\},\{1,2\}\}$ is not because $\{1\} \subseteq \{1,2\}$. □

DEFINITION 4.2.3. Minimization

μ is a unary operator that takes a set of sets S as input and it returns the minimal subset set $\mu(S)$ of S such that

- ❖ $\forall s_i, s_j \in \mu(S).\, s_i \subseteq s_j \leftrightarrow s_i = s_j$
- ❖ $\forall s' \in \mu(S).\exists s \in S.s' \subseteq s$
- ❖ $\forall s \in S.\, ((\neg\exists s' \in S.s' \subset s) \rightarrow s \in \mu(S))$ □

The first condition ensures that $\mu(S)$ is a minimal subset set. The second condition ensures that $\mu(S)$ does not contain any excess element, i.e., elements that are not subset of any set in S. The third condition ensures that $\mu(S)$ includes any element of S that is not proper subset of another element of S. All the three conditions together ensures the uniqueness of $\mu(S)$. For example, without the third condition, $\mu(\{\{1,2\},\{1,3\}\})$ could have been {}, {{}}, {{1}}, $\{\{1\},\{2\},\{3\}\}$ or $\{\{1\},\{3\}\}$.

If S contains s_1 and s_2 where $s_1 \subseteq s_2$, then only s_1 belongs to $\mu(S)$. The operator μ is introduced only to define the operators subset union $\oplus$ and subset product $\otimes$.

Definition 4.2.4. Subset union $\oplus$

The subset union is an n-ary commutative and associative operator that takes minimal subset sets as arguments and returns a minimal subset set. It is defined as follows:

- ❖ $S \oplus S = S$
- ❖ $S \oplus \emptyset = S$
- ❖ $S \oplus \perp = S$
- ❖ $S_1 \oplus S_2 \oplus ... \oplus S_n = \mu(S_1 \cup S_2 \cup ... \cup S_n)$ □

Example 4.2.5. Example applications of the subset union operator.

- ❖ $\{\{1\},\{2,3\}\} \oplus \{\{2,3,4\},\{5\}\} = \{\{1\},\{2,3\},\{5\}\}$
- ❖ $\{\{2,4\},\{3\}\} \oplus \{\{1\},\{2,3\}\} \oplus \{\{2,5\},\{4,5\},\{1,3\}\} = \{\{2,4\},\{3\},\{1\},\{2,5\},\{4,5\}\}$

□

Definition 4.2.6. Subset Product $\otimes$

The subset product $\otimes$ is an n-ary commutative and associative operator that takes minimal subset sets as arguments and returns a minimal subset set. It is defined as follows:

- $S \otimes S = S$
- $S \otimes \emptyset = S$
- $S \otimes \perp = \perp$
- $S_1 \otimes S_2 = \mu(S_1 \times S_2)$ where $A \times B = \{a \cup b : a \in A \wedge b \in B\}$
- $S_1 \otimes S_2 \otimes ... \otimes S_n = \mu(S_1 \times S_2 \times ... \times S_n)$ □

Example 4.2.7. Example applications of the subset product operator.

- $\{\{1\},\{2,3\}\} \otimes \{\{2,4\},\{3\}\} = \{\{1,2,4\},\{1,3\},\{2,3\}\} = \{\{2,4\},\{3\}\} \otimes \{\{1\},\{2,3\}\}$
- $\{\{2,4\},\{3\}\} \otimes \{\{1\},\{2,3\}\} \otimes \{\{2,5\},\{4,5\},\{1,3\}\} = \{\{1,2,4,5\},\{1,3\},\{2,3,5\}\}$

□

Definition 4.2.8. The condition graph

A node is a triple $< n, Excluded\,Conditions, Included\,Conditions >$ where

- n is either a single condition or a single rule. This implies that there are two kinds of nodes: condition nodes and rule nodes
- $Excluded\,Conditions$ and $Included\,Conditions$ are both sets of conditions

There is a special node $\perp$.

A *condition graph* is a quintuple $< N, root, C_0, \mathbb{C}, \rightarrow >$where

- N is a set of nodes
- *root* is the node $< g, \{g\}, \{\} >$ where g is the goal of the attacker. The root node is a condition node. For simplicity, we consider the case where the goal is a single condition. When there are several goal conditions $g_1, g_2, ..., g_k$, an extra rule $r_g : g_1, g_2, ..., g_k \rightarrow g$ will be added where the new goal condition will be g.
- C_0 is the set of initial conditions
- $\mathbb{C}$ is the set of reachable conditions computed in phase I
- $\rightarrow \subseteq N \times N$ is a reachability relation. There are three cases to be considered:
  - case 1. from a condition node to a rule node
    This relation describes how the condition c can be obtained. In particular, the descendant nodes of c correspond to rules that have c as their postcondition
    $n_c \rightarrow n_r$ where
    - n_c is a condition node $< c, X, I >$
    - n_r is a rule node $< r, X, I >$
    - c is the postcondition of r
    - $r.preconditions$ is the set of all the preconditions of r
    - $r.preconditions \subseteq \mathbb{C}$
  - case 2. from a rule node to a condition node
    This relation describes when a rule r is applicable. In particular, the descendant nodes of r correspond to the preconditions of r
    $n_r \rightarrow n_c$ where
    - n_r is a rule node $< r, X, I >$
    - n_c is a condition node $< c, X \cup \{c\}, I \cup r.preconditions \backslash \{c\} >$
    - c is a precondition of r
    - $c \notin X \cup C_0 \cup I$, the reason for introducing this condition will be discussed in Sect. 4.2.2

- ◇ case 3. from a rule node to a special node $\perp$
  $n_r \rightarrow \perp$ where
  - * n_r is a rule node $< r, X, I >$
  - * $\exists c \in r.preconditions.\ c \in X$

  This is the case where the current path in the condition graph represents a complex attack α that is covered by another complex attack α'. In this case, if R_a represents the set of rules that take part in a complex attack a, then $R_{\alpha'} \subset R_\alpha$. Therefore, there is no need to further explore this path. □

With respect to the multiple prerequisite graph [**12**] , the condition graph has a tree structure because all the paths are acyclic. The condition graph is not exhaustive [**22**] but it does cover all the complex attacks.

Example 4.2.9. An example condition graph. The configuration:

- ❖ the rules
  - ◇ $r_1 : 1 \rightarrow 2$
  - ◇ $r_2 : 2 \rightarrow 3$
  - ◇ $r_3 : 3 \rightarrow 6$
  - ◇ $r_4 : 1, 3 \rightarrow 4$
  - ◇ $r_5 : 4 \rightarrow 2$
  - ◇ $r_6 : 5 \rightarrow 4$
- ❖ the initial conditions are 1 and 5
- ❖ the goal condition is 6

The condition graph for this configuration is shown in Fig. 4.2.1. We have that $n_{r_4} \rightarrow \perp$ as $3 \in r_4.preconditions$ and $n_3 \in X_{n_{r_4}} = \{2, 3, 4, 6\}$ according to Def. 4.2.11. There is no need to further explore this path because, as we shall see in Sect. 4.2.2 , no condition needs to be obtained twice. Hence, for any complex attack α the contains two elementary attacks with the same postcondition, there exists another complex attacks α' which covers α. Formally, $\forall complex\ attack\ \alpha.$
$((\exists r_1, r_2 \in R_\alpha . r_1 \neq r_2 \wedge r_1.postcondition = r_2.postcondition) \rightarrow \exists \alpha'. R_{\alpha'} \subset R_\alpha)$

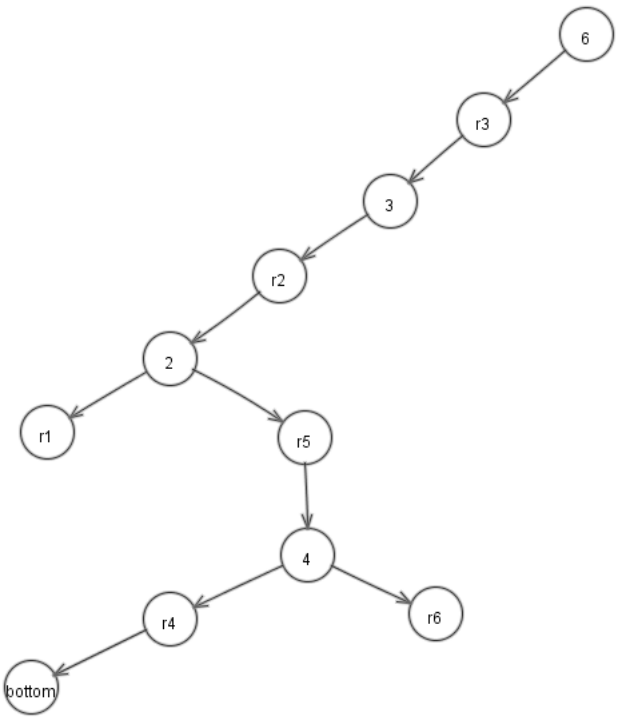


FIGURE 4.2.1. The condition graph.

Whereas $\perp$ is not a successor of n_{r_6} as the only precondition of r_6 is 1, which is not element of $X_{n_{r_6}} = \{2, 3, 4, 6\}$. The generated complex attack set Ω is $\{\{r_1, r_2, r_3\}\ ,\ \{r_6, r_5, r_2, r_3\}\}$. □

EXAMPLE 4.2.10. An example attack graph as described in Chapt.2 would generate the graph in Fig.4.2.2 with the same configuration as in Ex. 4.2.9. If we neglect cyclic paths, this attack graph contains the following complex attacks:

- ❖ r_1, r_2, r_3
- ❖ r_1, r_2, r_4, r_3
- ❖ r_1, r_2, r_6, r_3
- ❖ r_1, r_6, r_2, r_3
- ❖ r_6, r_1, r_2, r_3
- ❖ r_6, r_5, r_2, r_3

□

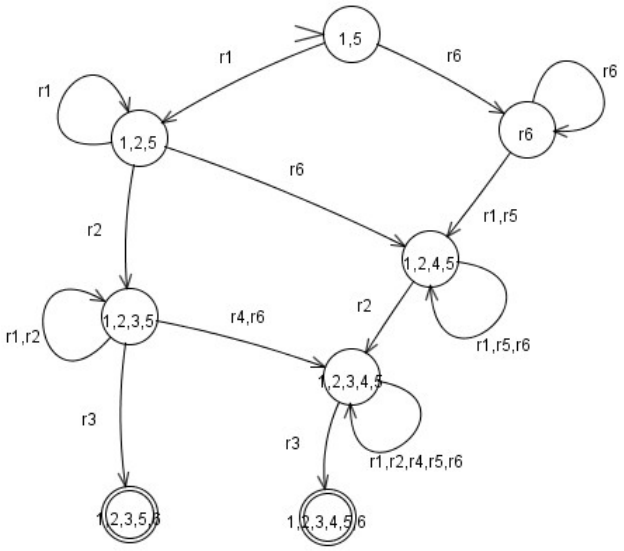


FIGURE 4.2.2. An example attack graph as described in Chapt.2.

We observe that for each complex attack α in Ex. 4.2.10, there exists an element in the complex attack set Ω generated through the condition graph that is a subset of α. Therefore, any set of countermeasures that stops all the complex attacks in Ω stops all the complex attacks generated through the attack graph as well.

The algorithm 4.2.16 computes the complex attack set Ω that covers all the complex attacks. It is a depth-first visit to the implicitly constructed condition graph. The complex attack set Ω is built from the graph by propagating upwards an output from each node, up to the root node, whose output will be the complex attack set Ω.

4.2.1. Output propagation

The output of a node is computed by applying the following rules:

- ❖ The output of a node without any out-going edge is the empty set {}

- ❖ The output of the special node $\perp$ is $\perp$[1]
- ❖ The output of a condition node n_c is $o_1 \oplus ... \oplus o_k$ where
  - ◇ $o_i = \{\{r_i\} \cup p \ : \ p \in n_i.output\}$ in the usual set builder notation. If $n_i.output = \{\}$ then $o_i = \{\{r_i\}\}$.
  - ◇ $n_1, ..., n_k$ are the successors of n_c
  - ◇ $n_i =< r_i, X, I >$ for $i = 1..k$
  - ◇ $n_i.output$ is the output of the node n_i , for $i = 1..k$
- ❖ The output of a rule node n_r is $n_1.output \otimes ... \otimes n_k.output$ where $n_1, ..., n_k$ are the successors of n_r.

Definition 4.2.11. Set of excluded conditions X

The set of excluded conditions X paired with a node n are the conditions represented by itself or by the direct/indirect parent nodes of n. This parameter is required to guarantee that for each rule node n_r, n_c is a descendant of n_r only if c is a precondition of r and that c does not belong to $n_r.X$. The set of excluded conditions of a node is inherited from those of its parent as follows:

- ❖ from a rule node to a condition node $n_r \rightarrow n_c : n_c.X = n_r.X \cup \{c\}$
- ❖ from a condition node to a rule node $n_c \rightarrow n_r : n_r.X = n_c.X$

where the notation $n.X$ denotes the set of excluded conditions paired with the node n. □

Example 4.2.12. The set of excluded conditions X paired with the node 4 in Fig. 4.2.1 is $\{2, 3, 4, 6\}$. □

The following definition closely resembles Def. 4.2.11.

[1]The first $\perp$ denotes the special node; the second $\perp$ denotes the special minimal subset set.

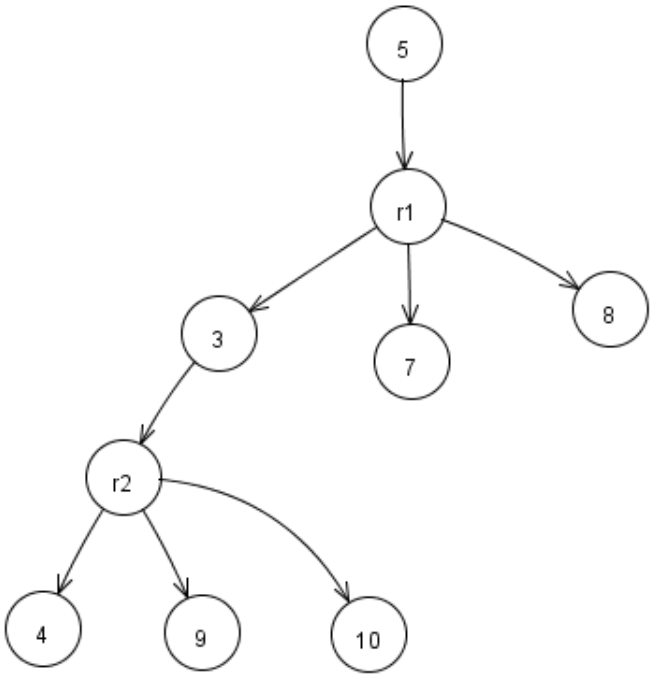


FIGURE 4.2.3.

DEFINITION 4.2.13. Set of included conditions I

The set of included conditions I paired with a node n are the conditions represented by the sibling nodes either of itself or of the direct/indirect parent nodes of n. This parameter is introduced to ensure that, for each condition node n_c, n_c is a descendant of a rule node n_r only if c does not belong to $n_r.I$. The set of included conditions of a node is inherited from those of its parent as follows:

- ❖ from a rule node to a condition node $n_r \rightarrow n_c : n_c.I = n_r.I \cup r.preconditions \backslash \{c\}$
- ❖ from a condition node to a rule node $n_c \rightarrow n_r : n_r.I = n_c.I$

where the notation $n.I$ denotes the set of included conditions paired with the node n. □

Example 4.2.14. The set of included conditions I paired with the condition node 4 in Fig. 4.2.3 is $\{7, 8, 9, 10\}$ where the conditions 9 and 10 are represented by the sibling nodes of the condition node 4 and the conditions 7 and 8 are represented by the sibling nodes of the ancestor condition node 3. □

The algorithm 4.2.16 generates the complex attack set Ω by

- ❖ visiting the condition graph
- ❖ building Ω in parallel

The creation of each *condition node* in the condition graph corresponds a recursive application of the function f .

Definition 4.2.15. Reachability index

A reachability index is a function that maps a condition c into the set of rules that have c as their postconditions. □

This index

- ❖ can be constructed in $O(|R|)$ time where $|R|$ is the cardinality of R
- ❖ does not change during the recursive application of algorithm

Algorithm 4.2.16. *Complex Attack Set Generation* ($goals$, C_0, $\mathbb{C}$, $index$)

1. $result \leftarrow \{\}$
2. *if* $(!goals \subseteq C_0 \wedge goals \subseteq \mathbb{C})$
3. *foreach* g *in* $goals$
4. $result \leftarrow result \otimes f(g, \{g\}, goals \backslash \{g\}, C_0, \mathbb{C}, index)$
5. *return* $result$

$f(c, X, I, C_0, \mathbb{C}, index)$

1. $result \leftarrow \{\}$
2. $predecessors \leftarrow index.get(c)$
3. *foreach* r *in* $predecessors$

4. $p \leftarrow r.preconditions \backslash (C_0 \cup I)$
5. *if* $p = \emptyset$
6. $result \leftarrow result \oplus \{\{r\}\}$
7. *else if* $(p \subseteq \mathbb{C} \wedge p \cap X = \emptyset)$
8. $subpaths \leftarrow \{\}$
9. *foreach* c' *in* p
10. $subpaths \leftarrow subpaths \otimes f(c', X \cup \{c'\}, I \cup p \backslash \{c'\}, C_0, \mathbb{C}, index)$
11. *if* $subpaths = \perp$
12. *break*
13. $result \leftarrow result \oplus \{\{r\} \cup e : e \in subpaths\}$
14. *if* $result = \{\}$
15. *return* $\perp$
16. *else*
17. *return* $result$

The algorithm inputs are:

- a set of reachable conditions $\mathbb{C}$ generated in phase I
- a set of initial conditions C_0
- a condition c, it corresponds to a condition node n_c in the condition graph
- a set of excluded conditions X paired with n_c
- a set of included conditions I paired with n_c
- a reachability index

The code for Complex Attack Set Generation operates as follows:

- the initial call to the function Complex Attack Set Generation creates the root node of the condition graph
- *result* is the output of the root node, that is, the complex attack set
- line 2 considers the following special cases:
  - the set of goals is a subset of the set of initial conditions
  - the set of goals contains an unreachable condition

  In either case, the algorithm terminates and returns the empty set. In the first case there is no set of countermeasures that can stop all the complex attacks. In the second case, instead, there is no feasible complex attack.
- line 3-4 compute the output of the root node. The descendant nodes of the root node n_g are condition nodes, each representing one of the subgoals.

The output of a descendant node n_c is the complex attack set to reach the subgoal c. The output of the root node is the subset product of the outputs of its descendant nodes.

f is an auxiliary function recursively defined. Each call to the function f creates a condition node n_c in the condition graph.

- ❖ line 1 initializes the output of n_c
- ❖ line 2 gets the list of rules that have c as their postconditions
- ❖ line 3-13 generate descendant rule nodes of n_c and update the output of n_c
- ❖ line 14-17 return the output of n_c. In particular, line 14-15 represent the special case discussed in Sect. 4.2.1
- ❖ each cycle of the for loop of line 3-13 creates a new rule node n_r
- ❖ line 5-6 represent the case where there is no out-going edges from the rule node n_r according to Def. 4.2.8, as the set of the preconditions of r is a subset of the union C_0 and I
- ❖ line 7-13 create descendant condition nodes of n_r by calling the function f recursively
- ❖ line 11-12 is the short-circuit evaluation of the operator subset product as defined in Def. 4.2.6. According to the definition in Sect. 4.2.1, the output of n_r is $n_1.output \otimes ... \otimes n_k.output$ where $n_1, ..., n_k$ are descendant nodes of n_r. This is the case when the output of one of the descendant nodes is $\perp$.
- ❖ line 13 computes the output of n_r

4.2.2. Correctness

The algorithm is correct if it returns a complete complex attack set. To begin with we observe two properties of a generic complex attack:

Property 1. No rule needs to be applied twice in a complex attack.

The existence of a complex attack path $a_1 \equiv s_0 \xrightarrow{r_1} s_1 \rightarrow \cdots \xrightarrow{r_i} s_i \rightarrow \cdots \rightarrow s_j \xrightarrow{r_i} s_{j+1} \rightarrow s_{j+2} \cdots \rightarrow s_{final}$, where the rule r_i is applied twice, implies that $s_j = s_{j+1}$ because of the monotonicity assumption. Therefore, any countermeasure that stops $a_2 \equiv s_0 \xrightarrow{r_1} s_1 \rightarrow \cdots \xrightarrow{r_i} s_i \rightarrow \cdots \rightarrow s_j \rightarrow s_{j+2} \rightarrow \cdots \rightarrow s_{final}$, which is the same as

a_1 except that r_i is applied once only, stops also a_1.

Property 2. No condition needs to be obtained twice.

This property

- ❖ is true in our case as rules are split as single postcondition rules
- ❖ does not hold in general, when a rule can have several postconditions as shown in Ex. 4.2.17.
- ❖ is a further consequence of monotonicity. If we group rules with the same postcondition together, at most one rule from each group is *necessary* for any complex attack. An example is shown in Ex. 4.2.18.

Example 4.2.17. This example shows that the property "no condition needs to be obtained twice" does not hold in general.
Rules:

- ❖ $r_1 : 1, 2 \rightarrow 3, 4$
- ❖ $r_2 : 6 \rightarrow 3, 5$
- ❖ $r_3 : 4, 5 \rightarrow 8$
- ❖ $r_4 : 7 \rightarrow 1, 6$

Initial conditions: $\{2, 7\}$
Goal : $\{8\}$

Possible complex attack sequences are $[r_4, r_2, r_1, r_3]$ and $[r_4, r_1, r_2, r_3]$. Both require more than one rule from the group of rules that have 3 in their postconditions. □

Example 4.2.18. This example explains why a complex attack requires at most one rule from each group.

Consider a complex attack a that contains two different rules r_1, r_2 with the same postcondition. a is defined as

$a \equiv s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_i \rightarrow s_{i+1} \rightarrow \cdots \rightarrow s_{final}$

a can be partitioned into two subpaths

$a' \equiv s_0 \rightarrow \cdots \rightarrow s_i$
$a" \equiv s_i \rightarrow s_{i+1} \rightarrow \cdots \rightarrow s_{final}$

such that $r_1 \in R_{a'}$ and $r_2 \in R_{a"}$ where $R_{a'}$ and $R_{a"}$ are the set of rules that take part in, respectively, a' and $a"$. The existence of a implies the existence of a complex attack $b \equiv s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_i \rightarrow s'_{i+1} \rightarrow \cdots \rightarrow s'_{final}$ and that b can be partitioned into two subpaths $b' \equiv s_0 \rightarrow \cdots \rightarrow s_i$ and $b" \equiv s_i \rightarrow s'_{i+1} \rightarrow \cdots \rightarrow s'_{final}$ such that:

- $a' = b'$
- $C_{a'}$ is the set of postconditions of the rules in $R_{a'}$
- $R_{b"}$ is the set of rules that take part in $b"$
- $C_{b"}$ is the set of postconditions of the rules in $R_{b"}$
- $C_{b"} \cap C_{a'} = \emptyset$
- $R_{b"} \subseteq R_{a"}$

Therefore, any countermeasure that stops the complex attack b also stops the complex attack a. □

The search of the Complex Attack Set Ω starts with the function call $Complex\ Attack\ Set\ Generation(goals, C_0, \mathbb{C}, index)$ where

- $goals$ is the set of goals of the attacker
- C_0 is the set of initial conditions

The condition graph would have contained all the attack paths as in the case of the goal-based backward graph if

- each condition node n_c had one outgoing edge to a rule node n_r for each rule r that has c as its postcondition
- each rule node n_r had one outgoing edge to a condition node n_c for each precondition c of r

Therefore, we consider the difference between the goal-based backward graph and the condition graph to understand why the latter covers all the attack paths. The

following cases are those that differ with respect to the goal-based backward graph.

Case 1. Condition node $n_c \equiv < c, X, I >$

There is one out-going edge for each rule that has c as its postcondition, except rules that have at least one precondition that does not belong to the set of reachable conditions $\mathbb{C}$. We observe that no complex attack is missed by omitting these exceptional rules because these rules are not going to be applicable anyway given that conditions that do not belong to $\mathbb{C}$ are not satisfiable.

Another consequence of excluding these rules is that any path explored in the condition graph corresponds to a viable path in the attack graph, where a viable path is a path from the initial state to a final state. The root node is added to the condition graph if the goal is a reachable condition. The restriction that a rule node n_r is added as a child of a condition node n_c only if the preconditions of r are all reachable ensures that conditions corresponding to the the children nodes of n_r are reachable. For example, we add the edge $c \rightarrow r_1$ to the graph in Fig. 4.2.4 only if all the preconditions of r_1 are reachable, which implies that $c_1, ..., c_n$ are reachable, i.e., there exists some paths in the attack graph $s_0 \rightarrow ... \rightarrow s_1$, ... , $s_0 \rightarrow ... \rightarrow s_n$ such that $c_1 \in s_1, ..., c_n \in s_n$. Under the hypothesis of monotonicity, these paths cannot be in conflict and therefore, there exists a path $s_0 \rightarrow ... \rightarrow s$ where $c_i \in s$, for $i = 1..n$. By further applying the rule r_1, the condition c will become valid. If c is the goal, then the path $s_0 \rightarrow ... \rightarrow s \xrightarrow{r_1} s_{final}$ is indeed a viable path.

If rules with unreachable conditions were not excluded, the output of the root node might contain a set that is not a subset of any complex attack, as shown in Ex. 4.2.19.

Case 2. Rule node $n_r \equiv < r, X, I >$

There is one out-going edge for each precondition of the rule r, with the exception of conditions that are elements of C_0, of X or of I .

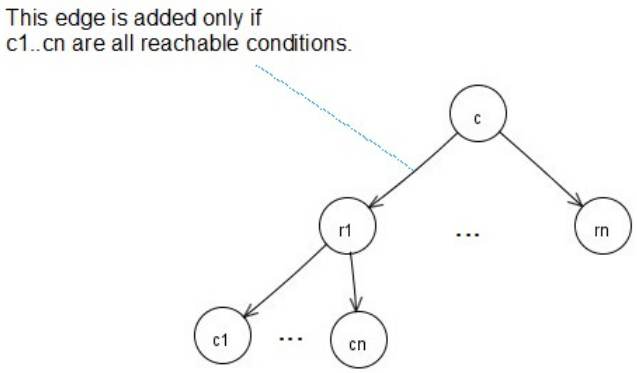


FIGURE 4.2.4.

(1) elements of C_0

For any complex attack α, a further application of any rule whose postcondition is a initial condition only results in a complex attack α' such that α is a subset of α'[2]. Therefore, there is no need to explore condition nodes in the condition graph that represent initial conditions.

(2) elements of X

Elements of X are conditions represented by itself or by some direct/indirect parents of the node n_r. Consider a generic condition c that is a precondition of r, the fact that c is in X implies that r has to be applied to satisfy c. In turn, this requires that c is satisfied. Therefore, the complex attack α represented by the path through n_r contains two distinct rules with the same postcondition c. As we observed that no condition needs to be obtained twice, the special node $\perp$ is assigned as a descendant of n_r to represent the fact that the current path does not need to be further explored, as shown in Ex. 4.2.20.

The special node $\perp$ denotes a path that yields a superfluous complex attack, this path is nonetheless viable. If we had not added $\perp$ as a successor of n_r, then the complex attack set Ω might contain a set s that increases the complexity of the search of the minimal set of countermeasures without contributing any useful information because:

[2] $\alpha \subseteq \alpha' \Leftrightarrow r \in \alpha \rightarrow r \in \alpha'$

(a) for any element s' in Ω different from s, any set of countermeasures that stops all the complex attacks in $\Omega \backslash \{s\}$ will stop all the complex attacks in Ω, as shown in Ex. 4.2.21;

(b) if the chosen countermeasure for s is not shared with any other set s' in Ω, then that countermeasure is not actually necessary and therefore, the computed set of countermeasures will not be minimal according to Def. 2.1.19.

(3) elements of I

Elements of I are conditions represented by siblings of itself or of direct successors of some direct/indirect parents of the node n_r. Suppose that:

(a) c is a precondition of r
(b) c is in I
(c) $n_{r'}$ is the common parent of n_r and n_c
(d) $c_1, ..., c_k, c$ are preconditions of r'.

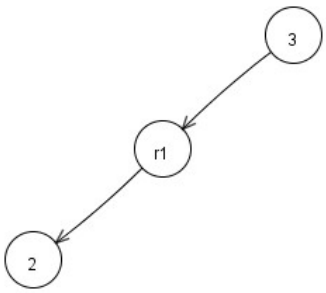
Then, the output of $n_{r'}$ is unchanged even if we do not add a successor node that represents the condition c to n_r because

- the output of $n_{r'}$ is given by $n_{c_1}.output \otimes ... \otimes n_{c_k.output} \otimes n_c.output$
- for any minimal subset set s, $s \otimes n_c.output = s \otimes n_c.output \otimes n_c.output$

Example 4.2.19. This example shows that the condition graph cannot include rules whose preconditions include unreachable conditions. Otherwise, the output of the root node may contain a set that is not a subset of any complex attack. The configuration is as follows:

- a single rule r where $r.preconditions = \{1, 2\}$ and $r.postcondition$ is 3
- $goal = \{3\}$
- $initial\, conditions = \{1\}$

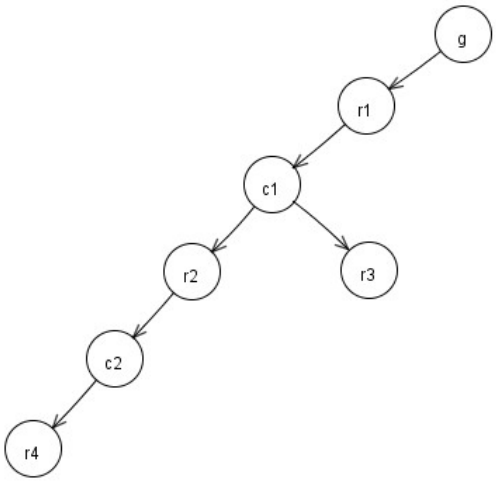
If rules that require unreachable conditions as their preconditions were not excluded, the condition graph would have been:



We have that

- $n_2.output = \{\}$
- $r_1.output = \{\}$
- $n_3.output = \{\{r_1\}\}$, but $\{r_1\}$ does not correspond to any complex attack since no attack is possible. □

Example 4.2.20. This example shows why a path that terminates with the special node $\perp$ does not need to be further explored.



The node relative to r_4 is $n_{r_4} =< r_4, \{1, 2, g\}, \{\} >$. Suppose that the rules are:

$r_1 : c_1 \rightarrow g$
$r_2 : c_2 \rightarrow c_1$
$r_3 : c_5 \rightarrow c_1$
$r_4 : c_1, c_7 \rightarrow c_2$

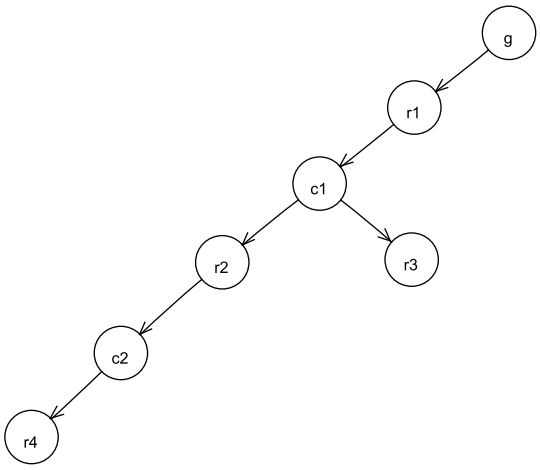
and $C_0 = \{c_5, c_7\}$.

The algorithm adds $\perp$ as a successor of n_{r_4} as $r_4.preconditions \cap \{1, 2\} \neq \emptyset$. We can see that the path r_1, r_2, r_4, r_3 is actually viable

$\{c_5, c_7\} \xrightarrow{r_3} \{c_1, c_5, c_7\} \xrightarrow{r_4} \{c_1, c_2, c_5, c_7\} \xrightarrow{r_2} \{c_1, c_2, c_5, c_7\} \xrightarrow{r_1} \{c_1, c_2, c_5, c_7, g\}$.

As r_2, r_3 have the same postcondition c_1, we know that there exists another complex attack α' which is a subset of $\{r_1, r_2, r_3, r_4\}$. In fact, the complex attack set Ω includes $\{r_1, r_3\}$ that covers $\{r_1, r_2, r_3, r_4\}$. Therefore, there is no need to further explore the node n_{r_4}. □

Example 4.2.21. This example shows that if the special node $\perp$ is not used, then the complex attack set may contain an excess element and, as a consequence, the set of countermeasures is no more minimal.



The configuration is as follows:

- ❖ the rules:
  $r_1 : c_1 \rightarrow g$
  $r_2 : c_2 \rightarrow c_1$
  $r_3 : c_3 \rightarrow c_1$
  $r_4 : c_1, c_7 \rightarrow c_2$
- ❖ initial conditions = $\{c_3, c_7\}$, goal = $\{g\}$

If the special node $\perp$ is *not* used, then we have:

$n_{r_4}.output = \{\}$
$n_{c_2}.output = \{\{r_4\}\}$
$n_{r_2}.output = \{\{r_4\}\}$
$n_{c_1}.output = \{\{r_2, r_4\}, \{r_3\}\}$
$n_{r_1}.output = \{\{r_2, r_4\}, \{r_3\}\}$
$n_g.output = \{\{r_1, r_2, r_4\}, \{r_1, r_3\}\}$

The complex attack covered by $\{r_1, r_2, r_4\}$ is $r_3 r_4 r_2 r_1$, which is also covered by $\{r_1, r_3\}$. Therefore, the set $\{r_1, r_2, r_4\}$ does not provide additional useful information to the complex attack set $\{\{r_1, r_3\}\}$. Furthermore, if the countermeasure c_{r_2} is chosen as the countermeasure for $\{r_1, r_2, r_4\}$ and the countermeasure c_{r_1} is chosen as the countermeasure for $\{r_1, r_3\}$, then the set of countermeasures $\{c_{r_2}, c_{r_1}\}$ is not minimal, because by eliminating c_{r_2} the set $\{c_{r_1}\}$ is still complete. □

4.2.3. Complexity

The worst case time complexity is $O(p \cdot m + |R|)$ where

- p is the number of complex attack paths in the attack graph;
- m is defined as $\underset{\alpha \in p}{max}|\alpha|$ where α is the complex attack path that involves the largest number of rules;
- $|R|$ is the total number of rules.

The complexity is output sensitive. As every path in the condition graph corresponds to at least one attack path in the attack graph and there are no two paths in the condition graph that correspond to the same attack sequence, therefore, $p \cdot m$ is the largest number of generated condition nodes.

4.3. Safeguard Selection

This section discusses the strategy to select countermeasures.

After computing the complex attack set Ω, the remaining problem to be considered is the selection of a set of countermeasures Σ that stops all the complex attacks. A set of countermeasures Σ stops all the complex attacks if and only if for each set ω in the complex attack set Ω, at least one element of ω belongs to the set of countermeasures Σ. This problem can be modeled as an integer linear programming problem (ILP). As proved in Sheyner *et al.* [**22**], the problem of choosing the minimal set of countermeasures is NP-complete. By modeling the problem as an ILP, we can leverage recent advances in integer programming methods. In particular, we can exploit the fact that we are interested in finding *a* solution, not necessarily the optimal one.

Even if the minimal set of countermeasures is complete, it is not always possible to adopt all those countermeasures under budgetary constraints. Therefore, a further selection of a subset is necessary such that the sum of their costs is within the budget. In order to stop all the complex attacks, the rules corresponding to countermeasures excluded from the subset should be inactivated via some alternative means. For example, by inserting a rule in the security policy that forbids the execution of an application. This subset selection problem can be modeled as a linear knapsack problem.

4.3.1. The Model

4.3.1.1. Decision variables

Let $x_1, ..., x_n$ be the decision variables where

- n is the total number of countermeasures
- $x_i \in \{0, 1\}$, $i = 1..n$
- $x_i = 0$ if the countermeasure associated with x_i is adopted; $x_i = 1$ if the countermeasure associated with x_i is not adopted

4.3.1.2. Objective function

The objective function to be minimized is defined as $\sum_{i=1..n} c_i(1 - x_i)$ where

- $x_1 ... x_n$ are the decision variables
- c_i is the cost of the countermeasure associated with x_i

While the interpretation of the cost depends upon the context, we are interested in minimizing the cost in any case. Some examples of the interpretation of the cost c_i may be:

- the cost of the implementation of the countermeasure
- the complexity of the implementation
- defined as $(1 - P(r_i))$ where
  - r_i is the rule associated with the countermeasure c_i
  - $P(r_i)$ is the probability of successful application of the rule r_i

    e.g., if r_i represents an attack, then $P(r_i)$ is the probability of the successful execution of that attack

4.3.1.3. Constraints

A constraint is introduced for each element ω of the complex attack set Ω. The constrains are defined as

$$\left(\sum_{i=1..n} M_{ji} \cdot x_i\right) \leq b_j - 1 \quad j = 1..|\Omega| \text{ where}$$

- M is the matrix representation of the complex attack set Ω where
  - each row of M corresponds to an element of Ω
  - each column of M corresponds to a countermeasure
  - $M_{ji} = 1$ if x_i is a countermeasure for the rule r where
    - $r \in \omega$

* ω corresponds to the row j of M

- ❖ $b_j = \sum_{i=1..n} M_{ji}$
- ❖ n is the total number of countermeasures
- ❖ $|\Omega|$ is the cardinality of Ω

As an example, the set $\{r_2, r_3, r_4\}$ in Ω generates the constraint $x_2 + x_3 + x_4 \leq 2$, which requires that at least one of x_2, x_3, x_4 has to be zero.

4.3.1.4. The complete model

$$minimize \quad \sum_{i=1..n} c_i(1 - x_i)$$

$$subject\,to \sum_{i=1..n} M_{ji} \cdot x_i \leq b_j - 1 \qquad j = 1..|\Omega|$$

$$x_i \in \{0, 1\} \qquad\qquad i = 1..n$$

EXAMPLE 4.3.1. This example shows how a model can be created from a complex attack set Ω. The example configuration is:

- ❖ the rules:
  - ◇ $r_1 :< \{c_1\}, \{c_2, c_3\} >$
  - ◇ $r_2 :< \{c_3\}, \{c_2, c_4\} >$
  - ◇ $r_3 :< \{c_3\}, \{c_8\} >$
  - ◇ $r_4 :< \{c_8\}, \{c_5\} >$
  - ◇ $r_5 :< \{c_1, c_4\}, \{c_5\} >$
  - ◇ $r_6 :< \{c_7\}, \{c_5\} >$
- ❖ the initial conditions: $\{c_1, c_3\}$
- ❖ the goal condition: $\{c_5\}$
- ❖ associations between the rules and the countermeasures:
  - ◇ $countermeasure_1$ is associated with r_1

- ◇ $countermeasure_2$ is associated with r_2
- ◇ $countermeasure_3$ is associated with r_3
- ◇ $countermeasure_4$ is associated with r_4
- ◇ $countermeasure_5$ is associated with r_5, r_6

❖ $cost_i$ is the cost of $countermeasure_i$, for $i = 1..5$

❖ $x_i = 0$ if $countermeasures_i$ is adopted, for $i = 1..5$

The above configuration will yield the complex attack set $\{\{r_2, r_5\}, \{r_3, r_4\}\}$
The problem of finding a minimal set of countermeasures can therefore be modeled as:

$$min \sum_{i=1..5} cost_i(1 - x_i)$$
$$x_2 + x_5 \leq 2 - 1$$
$$x_3 + x_4 \leq 2 - 1$$
$$x_i \in \{0, 1\} \;\; i = 1..5$$

□

Example 4.3.2. With reference to the example in Sect. 4.5, suppose that the countermeasure costs for the rules are defined as follows:

- ❖ r_1 : cost = 5
- ❖ r_2, r_5: cost = 4
- ❖ $r_3, r_4, r_6, .., r_{17}$: cost = 1
- ❖ r_{18}: cost = max

Then the problem of finding a minimal set of countermeasures may be modeled as

$$min \sum_{i=1..18} cost_i(1 - x_i)$$
$$x_1 + x_2 + x_5 + x_9 + x_{12} + x_{18} \leq 5$$
$$x_1 + x_2 + x_5 + x_{10} + x_{12} + x_{15} + x_{18} \leq 6$$
$$x_1 + x_2 + x_5 + x_9 + x_{13} + x_{15} + x_{18} \leq 6$$
$$x_1 + x_2 + x_5 + x_{10} + x_{13} + x_{15} + x_{18} \leq 6$$

$x_i \in \{0, 1\}$ $i = 1..18$

The computed minimal set of countermeasures for this problem is $\{x_{12}, x_{15}\}$, which means that we should inactivate r_{12} and r_{15}. □

4.3.2. Countermeasures within budget

After computing a minimal set of countermeasures for each threat profile, a subset of the minimal set is selected if

- ❖ there is a budget defined for that threat profile
- ❖ the total cost of countermeasures in the minimal set exceeds the budget

The problem of choosing this subset is modeled as a linear knapsack problem and is solved using dynamic programming in pseudo-polynomial time [**21**].

4.4. Complete example 1

This example is based on an example in Sheyner *et al.* [**22**].

As shown in Fig. 4.4.1, the network configuration includes three hosts IP_1, IP_2, IP_3. The intruder launches attacks from the host IP_1. The instantiated conditions are

- ❖ c_1, c_2, c_3: user-level privileges on IP_1, IP_2, IP_3, respectively
- ❖ c_4, c_5, c_6: root-level privileges on IP_1, IP_2, IP_3, respectively
- ❖ c_{11} : *sshd* running on IP_2
- ❖ c_{12}, c_{13}: the presence of ftp vulnerabilities on IP_2, IP_3, respectively
- ❖ c_{15}, c_{16} : the presence of buffer overflow vulnerabilities on IP_2, IP_3, respectively
- ❖ c_{32}, c_{33}, c_{34}: remote login relationship between IP_1 and IP_2, IP_1 and IP_3, IP_2 and IP_3, respectively

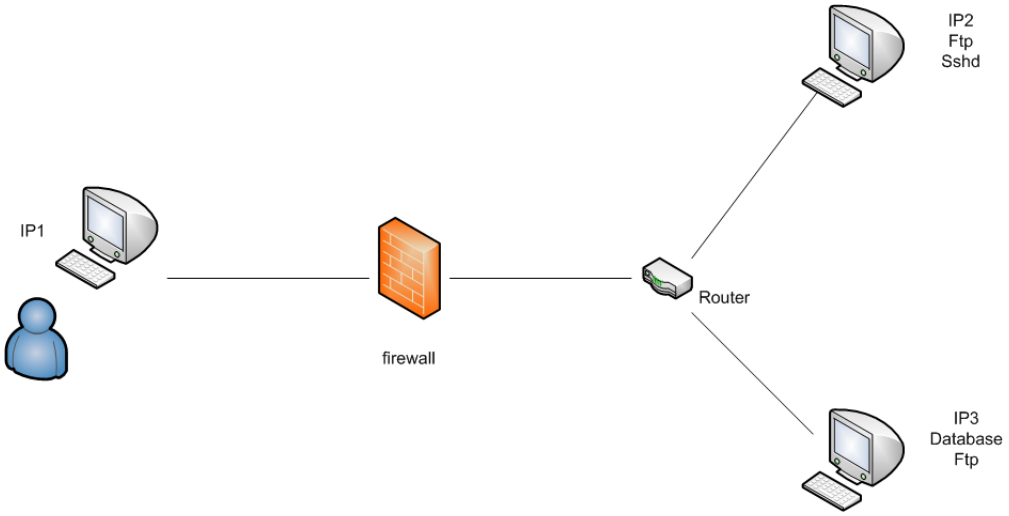


FIGURE 4.4.1. The example network configuration

- c_{21}: possibility to connect to IP_2 on the sshd port

The set of initial conditions C_0 is $\{c_4, c_{12}, c_{21}, c_{13}, c_{16}, c_{11}\}$. The goal of the attacker is to gain root-level privileges on IP_3 that would enable him to compromise the database, therefore, *goal* $=\{c_6\}$. The rules represent:

- dependencies that express that whoever has root-level privileges on a host has also user-level privileges on that host.
  - $r_1 : c_4 \rightarrow c_1$
  - $r_2 : c_5 \rightarrow c_2$
- the *sshd buffer overflow attack* whose preconditions are the presence of ftp vulnerability on a host t, user-level privileges on a host a, the possibility to connect to t on the sshd port. The postcondition is root-level privileges on t.
  - $r_3 : c_1, c_{11}, c_{21} \rightarrow c_5$
  - $r_4 : c_3, c_{11}, c_{21} \rightarrow c_5$
- the *ftp .rhosts attack* whose preconditions are the presence of the ftp vulnerability on a host t and user-level privileges on a host a. The postcondition is the establishment of a remote login relationship between a and t.
  - $r_5 : c_1, c_{13} \rightarrow c_{33}$
  - $r_6 : c_1, c_{12} \rightarrow c_{32}$

- $r_7 : c_2, c_{13} \rightarrow c_{34}$
- $r_8 : c_3, c_{12} \rightarrow c_{34}$

❖ the *remote login attack* which uses an existing remote login trust relationship between two hosts to gain user-level privileges on the other host without supplying a password.

- $r_9 : c_{32} \rightarrow c_2$
- $r_{10} : c_{33} \rightarrow c_3$
- $r_{11} : c_{34} \rightarrow c_3$

❖ the local buffer overflow attack whose preconditions are user-level privileges on host t and the presence of buffer overflow vulnerability on host t . The postcondition is root-level privileges on t.

- $r_{12} : c_2, c_{15} \rightarrow c_5$
- $r_{13} : c_3, c_{16} \rightarrow c_6$

The set of reachable conditions $\mathbb{C}$ that is generated by applying the algorithm 4.1.1 is $\{c_1, c_2, c_3, c_4, c_5, c_6, c_{11}, c_{12}, c_{13}, c_{16}, c_{21}, c_{32}, c_{33}, c_{34}\}$. The condition graph implicitly visited by the algorithm is shown in Fig.4.4.3. The complex attack set that is computed is $\{\{r_1, r_5, r_{10}, r_{13}\}, \{r_1, r_2, r_3, r_7, r_{11}, r_{13}\}, \{r_1, r_6, r_7, r_9, r_{11}, r_{13}\}\}$.

Fig.4.4.4 shows a partial attack graph for the same configuration. The number of condition nodes in the condition graph is much lower than the number of states in the attack graph. This is due to the lower amount of information contained in the condition graph. In particular, paths in the condition graph are not complex attack sequences as in the case of an attack graph, as shown in Ex. 4.4.1.

Example 4.4.1. This example shows that paths in the condition graph are not complex attack sequences, instead they represent sets of rules that *cover* some complex attacks. The configuration is as follows:

❖ the rules:

- $r_1 : c_2, c_3, c_4 \rightarrow c_g$
- $r_2 : c_3, c_4 \rightarrow c_2$
- $r_3 : c_{10} \rightarrow c_3$
- $r_4 : c_{11} \rightarrow c_4$

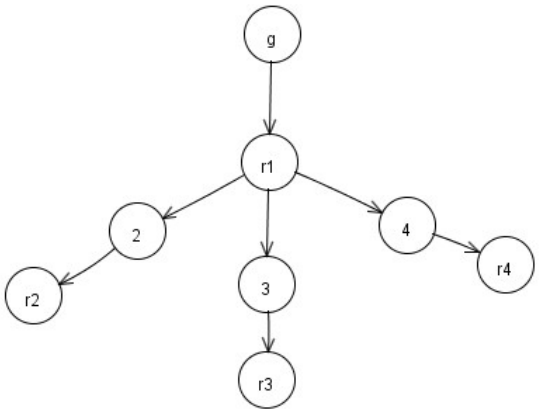


FIGURE 4.4.2. The condition graph

❖ the initial conditions are c_{10} and c_{11}; the goal is c_g

The condition graph is shown in Fig. 4.4.2. The path $c_g \rightarrow r_1 \rightarrow c_2 \rightarrow r_2$ is not a complex attack sequence because the goal cannot be reached by applying r_1 and r_2 alone. However, the path does *cover* the complex attack r_1, r_2, r_3, r_4 because $\{r_1, r_2\} \subseteq \{r_1, r_2, r_3, r_4\}$. □

4.5. Complete example 2

The previous example has shown that the condition graph is much smaller than the attack graph. This example will show the advantage of the condition graph over the goal-based backward graph and it is based upon an example in Sheyner and Wing [**23**]. The network configuration is shown in Fig. 4.5.1.

There are four hosts IP_1, IP_2, IP_3, IP_4. The intruder launches attacks from the host IP_1. The instantiated conditions are defined as follows:

Connectivity related:

❖ $c_{14}, c_{15}, c_{16}, c_{17}, c_{18}, c_{19}, c_{20}, c_{21}, c_{22}, c_{23}, c_{24}, c_{25}$: physical connectivity between hosts IP_1 and IP_1 , IP_1 and IP_2 , IP_2 and IP_1, IP_2 and IP_2, IP_2 and IP_3,

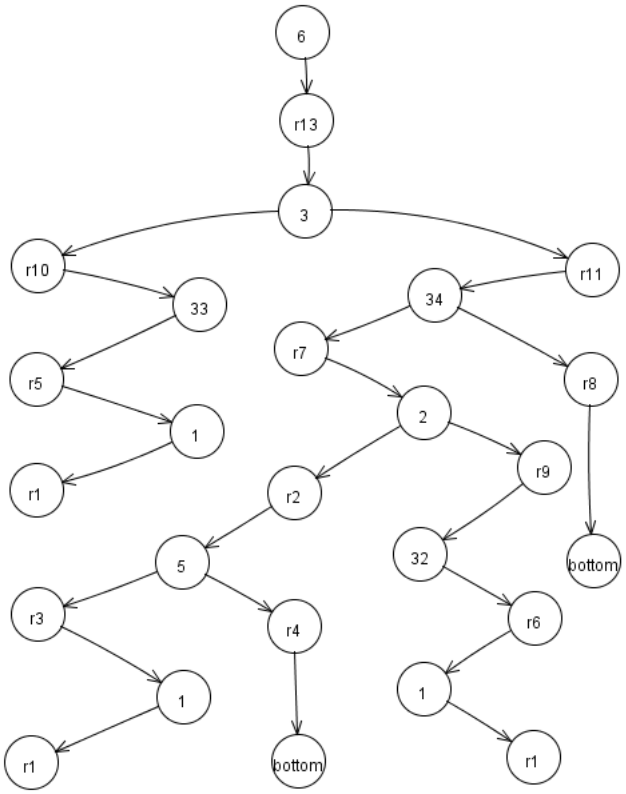


FIGURE 4.4.3. The complete condition graph

IP_2 and IP_4, IP_3 and IP_2, IP_3 and IP_3, IP_3 and IP_4, IP_4 and IP_2, IP_4 and IP_3, IP_4 and IP_4, respectively

- $c_{26}, c_{27}, c_{28}, c_{29}, c_{30}, c_{31}, c_{32}, c_{33}, c_{34}, c_{35}, c_{36}$: the ability to connect to port 80 between hosts IP_1 and IP_1, IP_1 and IP_2, IP_2 and IP_2, IP_2 and IP_3 , IP_2 and IP_4, IP_3 and IP_2, IP_3 and IP_3, IP_3 and IP_4, IP_4 and IP_2, IP_4 and IP_3, IP_4 and IP_4, respectively

- $c_{37} - c_{44}$: the ability to connect to port 5190 between hosts IP_2 and IP_3, IP_2 and IP_4, IP_3 and IP_4, IP_4 and IP_3, and between hosts IP_i and IP_i for $i = 1..4$

Privileges related:

- c_8, c_7, c_6, c_{53}: root-level privileges on host IP_1, IP_2, IP_4, IP_3, respectively

- $c_9, c_{10}, c_{11}, c_{12}$: user-level privileges on host IP_2, IP_4, IP_1, IP_3, respectively

Services related and others:

- ❖ c_1: a vulnerable version of IIS web service running on IP_2
- ❖ c_2 : a vulnerable version of Squid proxy running on IP_4
- ❖ c_3 : a vulnerable version of Licq running on IP_4
- ❖ c_4: HTML scripting is enabled on IP_3
- ❖ c_5: *at* executable vulnerable to overflow on IP_4
- ❖ c_{13} : the intruder has performed a port scan on the target network

The set of initial conditions C_0 is $\{c_1, c_2, c_3, c_4, c_5, c_8\} \cup \{c_i : i = 14..44\}$. The goal of the intruder is to gain root-level privileges on IP_4 to compromise the database, therefore, $goal = \{c_6\}$. The rules describe:

- ❖ dependencies expressing that whoever has root-level privileges on a host has also user-level privileges on that host
  - ◇ $r_1 : c_8 \rightarrow c_{11}$
  - ◇ $r_2 : c_7 \rightarrow c_9$
  - ◇ $r_3 : c_6 \rightarrow c_{10}$
  - ◇ $r_4 : c_{53} \rightarrow c_{12}$
- ❖ the IIS buffer overflow attack. The attack preconditions are user-level privileges on a host s, that a vulnerable version of IIS web service is running on IP_2 and that IP_2 is reachable from s on port 80. The attack postcondition is root-level privileges on IP_2.
  - ◇ $r_5 : c_{11}, c_1, c_{27} \rightarrow c_7$
  - ◇ $r_6 : c_9, c_1, c_{28} \rightarrow c_7$
  - ◇ $r_7 : c_{12}, c_1, c_{31} \rightarrow c_7$
  - ◇ $r_8 : c_{10}, c_1, c_{34} \rightarrow c_7$
- ❖ the Squid port scan attack whose preconditions are user-level privileges on a host s, that a vulnerable version of Squid proxy is running on IP_4 and that IP_4 is reachable from s on port 80. The attack postcondition is the condition that the intruder has performed a port scan on the target network.
  - ◇ $r_9 : c_{30}, c_9, c_2 \rightarrow c_{13}$
  - ◇ $r_{10} : c_{33}, c_{12}, c_2 \rightarrow c_{13}$

- $r_{11} : c_{36}, c_{10}, c_2 \rightarrow c_{13}$

❖ the Licq remote to user attack. The attack preconditions are user-level privileges on a host s, that a vulnerable version of Licq is running on IP_4, that IP_4 is reachable from s on port 5190 and that the intruder has performed a port scan on the target network. The attack postcondition is user-level privileges on IP_4.

- $r_{12} : c_{40}, c_9, c_{13}, c_3 \rightarrow c_{10}$
- $r_{13} : c_{42}, c_{12}, c_{13}, c_3 \rightarrow c_{10}$
- $r_{14} : c_{44}, c_{10}, c_{13}, c_3 \rightarrow c_{10}$

❖ the remote-to-user attack. The attack preconditions are user-level privileges on a host s, that HTML scripting is enabled on IP_3 and that IP_3 is reachable from s on port 80. The attack postcondition is user-level privileges on IP_3.

- $r_{15} : c_{31}, c_9, c_4 \rightarrow c_{12}$
- $r_{16} : c_{32}, c_{12}, c_4 \rightarrow c_{12}$
- $r_{17} : c_{33}, c_{10}, c_4 \rightarrow c_{12}$

❖ the local buffer overflow attack. The attack preconditions are user-level privileges on a host IP_4 and that a vulnerable *at* executable is running on IP_4. The attack postcondition is root-level privileges on IP_4.

- $r_{18} : c_{10}, c_5 \rightarrow c_6$

The set of reachable conditions $\mathbb{C}$ is generated in six iterations by applying the algorithm in Fig. 4.1.1. The sets of reachable conditions generated in each iteration are shown below:

$\{c_1, c_2, c_3, c_4, c_5, c_8, c_{14}, ..., c_{44}\}$

$\xrightarrow{r_1} \{c_1, c_2, c_3, c_4, c_5, c_7, c_8, c_{14}, ..., c_{44}\}$

$\xrightarrow{r_2,r_5} \{c_1, c_2, c_3, c_4, c_5, c_7, c_8, c_9, c_{11}, c_{14}, ..., c_{44}\}$

$\xrightarrow{r_6,r_9,r_{15}} \{c_1, c_2, c_3, c_4, c_5, c_7, c_8, c_9, c_{11}, c_{12}, c_{13}, c_{14}, ..., c_{44}\}$

$\xrightarrow{r_7,r_{10},r_{12},r_{13},r_{14},r_{16}} \{c_1, c_2, c_3, c_4, c_5, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, ..., c_{44}\}$

$\xrightarrow{r_8,r_{11},r_{17},r_{18}} \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, ..., c_{44}\}$

$\xrightarrow{r_3} \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, ..., c_{44}\}$

The condition graph implicitly constructed and visited by the algorithm is shown in Fig. 4.5.2. The shaded nodes corresponds to recursive calls to the function f, sixteen condition nodes are generated in this example. The computed Complex Attack Set is $\{\{r_1, r_2, r_5, r_9, r_{12}, r_{18}\}, \{r_1, r_2, r_5, r_{10}, r_{12}, r_{15}, r_{18}\}$, $\{r_1, r_2, r_5, r_9, r_{13}, r_{15}, r_{18}\}, \{r_1, r_2, r_5, r_{10}, r_{13}, r_{15}, r_{18}\}\}$.

The complete goal-based backward graph for the same network is too large to be shown here. We describe some of the pruned branches of the condition graph with respect to the goal-based backward graph.

- ❖ the node n_1 would have the subtree rooted at n_2 as its successor
- ❖ the node n_4 would have the subtree rooted at n_3 as its successor
- ❖ the node n_6 would have the subtree rooted at n_5 as its successor
- ❖ the node n_8 would have the subtree rooted at n_7 as its successor

The graphical representation of the condition graph is not part of the final output as it does not help comprehension of attack scenarios because paths in the condition graph are not complex attack sequences.

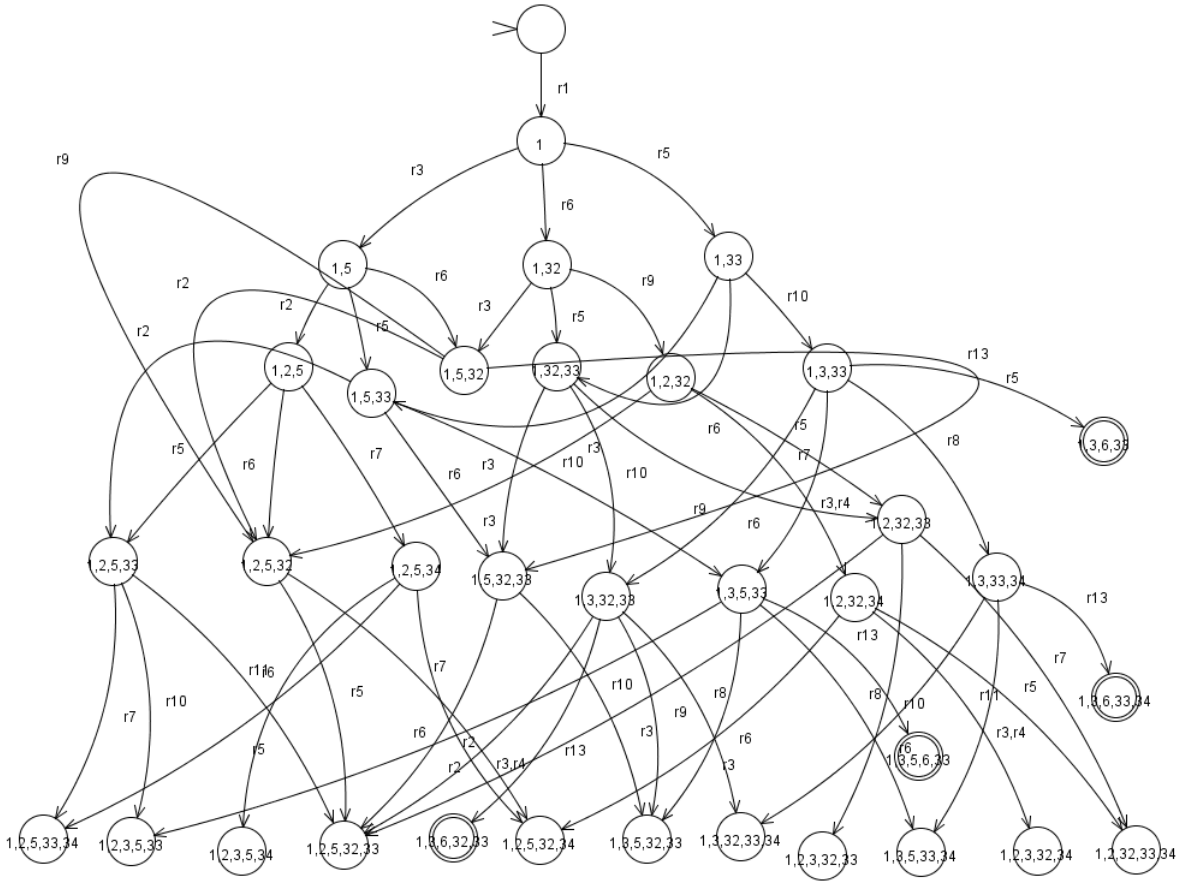


FIGURE 4.4.4. Part of the attack graph (the nodes at the bottom-most level are still to be expanded)

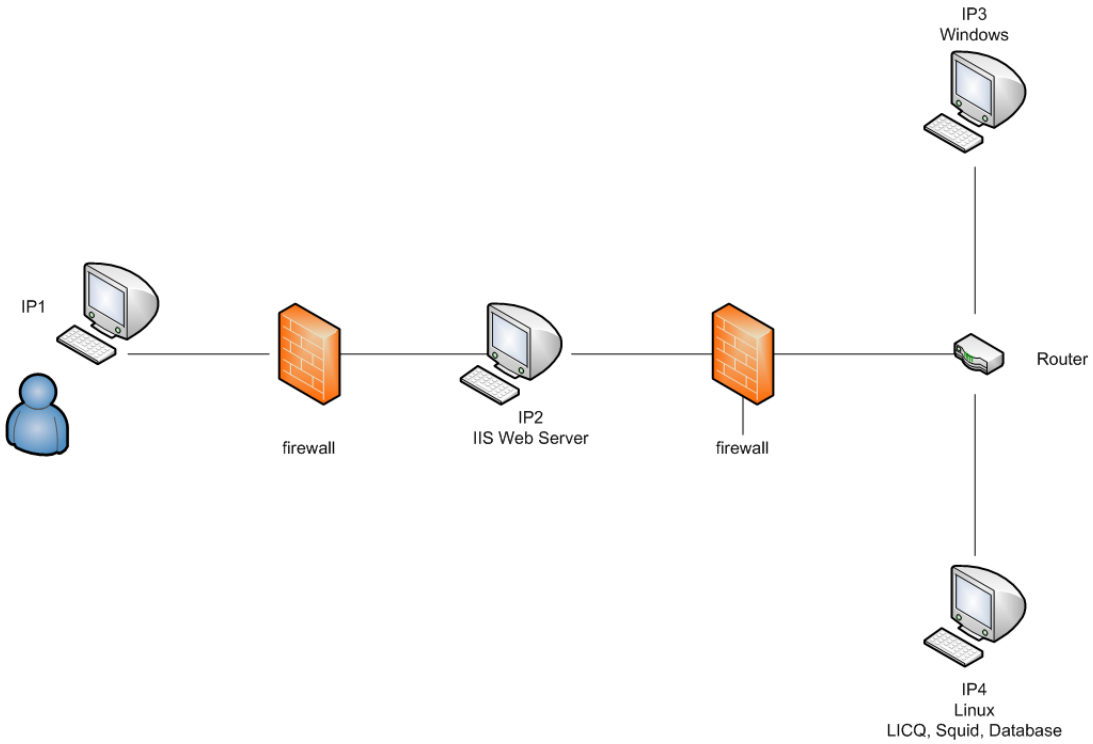


FIGURE 4.5.1. The example network configuration

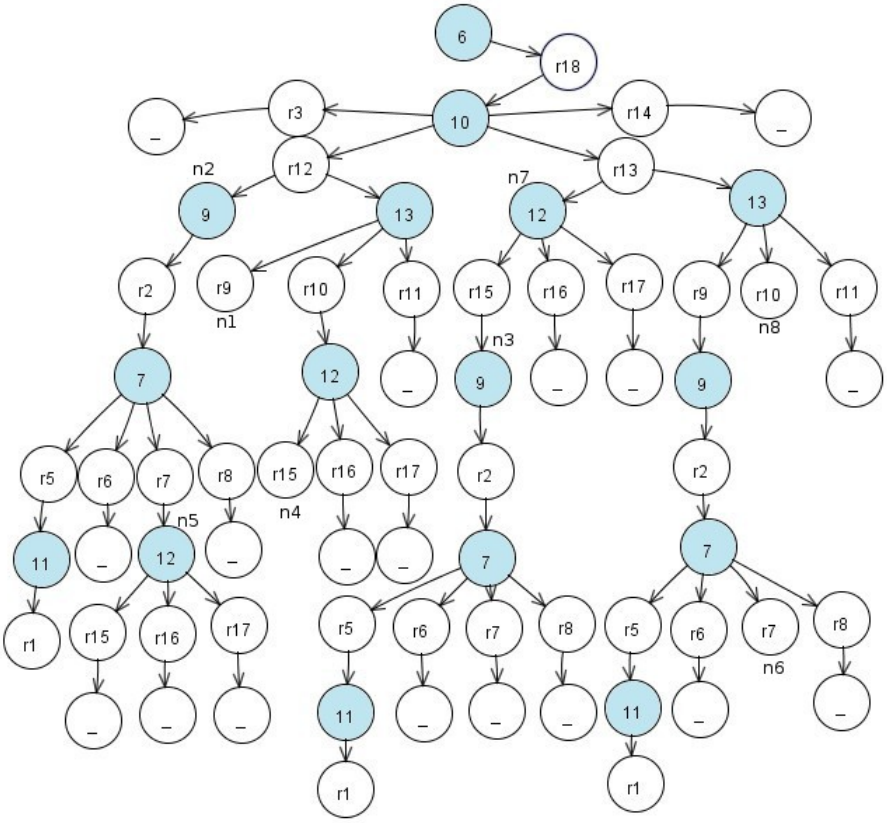


FIGURE 4.5.2. The condition graph, the nodes labeled "_" are the special nodes $\perp$.

Chapter 5

Attack Graph Toolkit

We have implemented a toolkit to help defending large networks from intelligent threats by suggesting a minimal set of countermeasures, whose application would stop the threats from achieving their goals. An threat is intelligent if it is goal-oriented [**10**]. We assume that the result of the threat analysis of the infrastructure is available. In particular, both the goals of the intelligent threats and the resources at their disposal are known. The toolkit has been designed to be used as a backend in combination with a frontend and the overall architecture is shown in Fig. 5.0.1.

The frontend analyzes a network and collects information from vulnerability scanners, firewall rules and vulnerability databases. It should also provide a user interface where users specify connectivities of the network and dependencies among the network components. The connectivities and the dependencies should be relatively static in time whereas the vulnerability information is more dynamic in nature but it is more readily collectible automatically. After collecting the input data, the most complex part is the enumeration of conditions, which depends upon the chosen detail level. For example, the condition "write access to the folder /usr/local/share on host A" can be substituted by "control over the integrity of A" when modeling at a coarser grain level. The implementation of the frontend is out of the scope of this thesis. Alternative implementations of automatic generation of input data are explored in [**23, 12**].

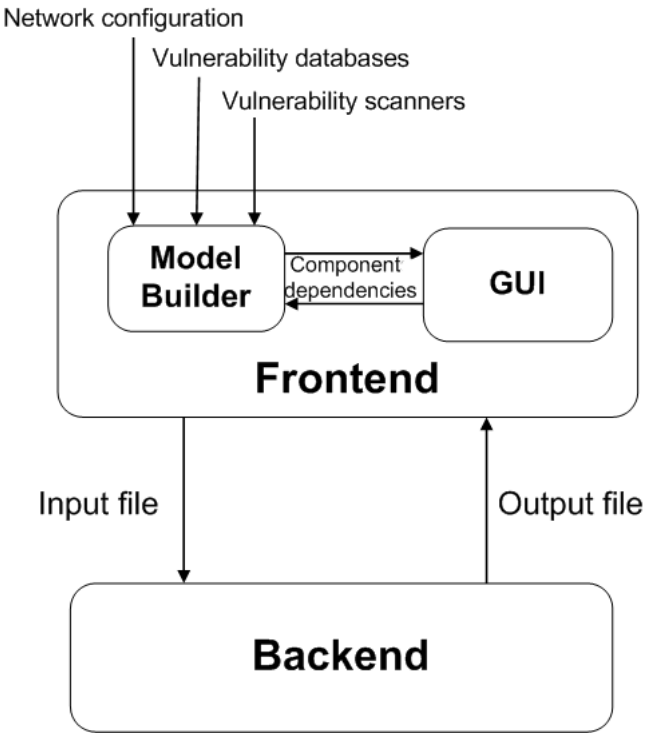


FIGURE 5.0.1. The overall architecture

The infrastructure model is specified in the input file and it describes dependencies, vulnerability instances of network components and threat profiles. A threat profile is relative to a hypothetical adversary and consists of the identification of the threat profile, the goals and the initial conditions regarding that adversary. For each threat profile, the backend generates a Complex Attack Set and a minimal set of countermeasures. The backend is decomposed into three components:

(1) the Input Parser
(2) the Complex Attack Set Generator
(3) the Countermeasure Selector

The interactions between the three components are shown in Fig. 5.0.2. The backend is implemented in Java and the three components are structured as three packages. The input file and the output file define the interface between the backend and the frontend.

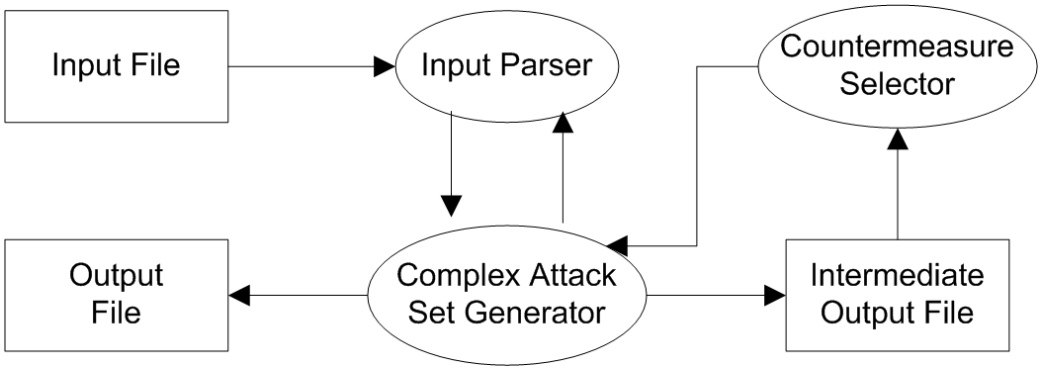


FIGURE 5.0.2. The interactions between the Input Parser, the Complex Attack Set Generator and the Countermeasures Selector

5.1. THE INPUT FILE

The input file describes the infrastructure model, which is expressed through:

- ❖ rules that describe connectivities of the network, dependencies among the network components and vulnerability instances of the network components. Each rule is characterized by a set of preconditions and a set of postconditions as described in Sect. 2.1.
- ❖ threat profiles, each profile is characterized by:
  - ◇ an identification of the profile
  - ◇ a set of goal conditions
  - ◇ a set of initial conditions

5.1.1. Format of the input file

The input file is a text file. Informally, each line can be

- ❖ a comment: a comment can start anywhere in a line and is followed by a double-slash
- ❖ a rule: a rule is specified as *preconditions* → *postconditions* where
  - ◇ *preconditions* is a list of *zero* or more conditions, separated by commas if there are several conditions

◇ *postconditions* is a list of *one* or more conditions, separated by commas if there are several conditions

◇ a rule can optionally be prefixed by:

* a non-negative real number within parentheses that represents the cost of the countermeasure relative to that rule.
* the symbolic value *max* within parentheses that implies that there is no countermeasure available for that rule. Therefore, if the recommended set of countermeasures includes a countermeasure whose cost is *max*, no set of countermeasures can stop all the complex attacks.

When a rule is not prefixed by neither a real number nor the value *max*, then the default cost applies. This cost is specified as a variable. When the default cost variable is undefined, the value 1.0 is used. Some example rules are shown in Ex. 5.1.1

❖ a list of rules: more than one rule may be listed on the same line, separated by semicolons. A single countermeasure, which is associated with all the rules listed on the same line, stops all or none of the rules.

❖ a definition of a variable: any line prefixed by # defines a variable. There are four kinds of variables

(1) default countermeasure cost: it is specified as
$\#default = RealNumber$ where $RealNumber$ can be any non negative real number.
e.g., $\#default = 2.3$
If the input file does not define the default cost, then the cost 1 is used.

(2) goal conditions: The set of goal conditions can be specified as
$\#goal(Name) = Conditions$ where
   (a) $Name$ is an identifier of a threat profile
   (b) $Conditions$ is a list of one or more conditions, separated by commas and enclosed in curly braces

(3) initial conditions: The set of initial conditions can be specified as
$\#init(Name) = Conditions$ where
   (a) $Name$ is an identifier of a threat profile

(b) *Conditions* is a list of zero or more conditions, separated by commas and enclosed in curly braces

(4) budgets: The budget for countermeasures can be specified as $\#budget(Name) = RealNumber$ where
   (a) $RealNumber$ can be any non negative real number
   (b) $Name$ is an identifier of a threat profile

   When this variable is undefined for a threat profile, the budget is assumed to be infinite.

For each threat profile, both a set of initial conditions and a set of goal conditions have to be defined. If the set of initial conditions is missing for a threat profile, then that profile is not considered. If the set of goal conditions is missing for a threat profile, an error is signaled.

When the same variable is defined more than once, the last definition will override the previous ones. The redefined definitions will be logged.

Unrecognized lines and white spaces are ignored by the parser. The grammar of the input file language and an example input file can be found in Appendix A.

Example 5.1.1. Example rules in an input file:

- $1, 2 \rightarrow 3$ represents a rule with the conditions enumerated 1 and 2 as preconditions and the condition enumerated 3 as postcondition. The cost of the corresponding countermeasure is the default one.

- $(2.24)\, 3 \rightarrow 2$ represents a rule with the condition enumerated 3 as precondition and the condition enumerated 2 as postcondition. The cost of the countermeasure for this rule is 2.24.

- $(max)\, 1 \rightarrow 2$ represents a rule that does not have a countermeasure. □

5.2. The Output File

The output file lists the minimal set of countermeasures for each threat profile.

5.2.1. Format of the output file

The output file is a *copy* of the input file, but where:

(1) rule(s) lines are enumerated, where the enumeration of a line represents the identification of the countermeasure associated with the rule(s) of that line.

(2) there is a new variable for each threat profile, defined as $\#countermeasure(Name) = Countermeasures$ where
- $Name$ is the identifier of a threat profile
- $Countermeasures$ is a list of the zero or more identifications of countermeasures, separated by commas and enclosed in curly braces.

(3) there is a new variable $countermeasuresWithinBudget$ for a threat profile p if the total cost of countermeasures exceeds the budget relative to p. This variable is defined as
$\#countermeasuresWithinBudget(Name) = Countermeasures$

The grammar of the output file language can be found in Appendix A.2.

5.3. The Input Parser

The input parser sequentially parses the input file line by line through the use of regular expressions and thereby implicitly adopts the default-deny policy discarding illegal lines. The skeleton of the output file is created during the input file parsing, because most of the output file is just a verbatim copy of the input one. The output file will be finalized in a latter stage, when the minimal set of countermeasures for each threat profile is computed. The input parser is the only interface between the input file and the other components. Therefore, should the input language be changed, only the input parser needs to be modified.

5.3.1. Data structures

(1) Conditions
As we have assumed that all the conditions are enumerated, each condition is represented as an integer.

(2) A Rule
A *Rule* is a record with three fields:
- *id*, an integer. When a rule with n postconditions is split into n rules, all of these rules will share the same id.
- *preconditions*, an array of integers.
- *postcondition*, an integer.

(3) Rules
Rules is an abstract data type that represents a set of rules, internally it is implemented as a linked list of *Rule*.

(4) Cost of countermeasures
The variable *countermeasureCost* is a vector of *Double* . The $(i-1)^{th}$ element is the cost of the rule identified by the id i. When the countermeasure cost of a rule is the symbolic value max, the corresponding element of *countermeasureCost* is *null*.

5.3.2. The interface

After parsing the input file, the input parser stores the data in its internal format in the data structures previously described. Subsequent interactions with the input parser depend upon the memorized data, any modification to the input file from this stage on does not affect the computation of the minimal set of countermeasures. Input Parser interfaces with other components of the backend via following functions:

- $int[]\, getGoal\, (String\ id)$ returns an array of integers that represents the goal conditions of the threat profile identified by *id*.
- $Double\, getSumOfCountermeasureCosts\, ()$ returns the sum of the costs of all countermeasures, this sum is needed to formulate the problem of finding

a minimal set of countermeasures as an ILP problem. In particular, the symbolic value max equals to the value $m + 1$ where m is the sum of the costs of all countermeasures.

- $int[\,]\, getInitCond(String\ id)$ returns an array of integers that represents the initial conditions of the threat profile identified by id.

- $int\, getNumOfRules()$ returns the total number of rules in the input file. Though not necessary, this value simplifies the generation of the ILP problem in the CPLEX LP format.

- $Set < String > \ getNames()$ returns a set of strings which are identifiers of the threat profiles.

- $Double[\,]\, getCountermeasureCost()$ returns the $countermeasureCost$ vector converted as an array.

- $Rules\, getRules()$ returns the set of rules represented in the abstract data type $Rules$.

- $Double\, getBudget(String\, id)$ returns the budget relative to the threat profile identified by id.

5.4. The Complex Attack Set Generator

The complex attack set generator is the core component of the backend. It communicates with the input parser to get the input data and it generates a complex attack set for each threat profile. Then, it formulates the problem of finding a minimal set of countermeasures for each complex attack set as an integer linear programming problem (ILP). The formulated problem is saved in an intermediate output file in the CPLEX LP format [**2**] .

The complex attack set generator implements the algorithms described in Chapt. 4 to generate a complex attack set. Then the set is passed to the countermeasure selector.

5.4.1. Data structures

(1) Reachability Index
The variable *reachabilityIndex* of type *Index* is a function that maps a condition c into a set of rules, which have c as their postcondition. This data structure can be statically computed. The time and space complexity are linear in the number of rules. The type *Index* is implemented through a hashtable.

(2) Attackability
The variable *attackability* is a function that maps a string into an integer. The string identifies a threat profile, whereas the integer is used as an enum and it can assume one of the three possible values:

- $NO\_ATTACK\_POSSIBLE$
- $NO\_POSSIBLE\_COUNTERMEASURE$
- $ATTACK\_POSSIBLE$

$NO\_ATTACK\_POSSIBLE$ denotes that, given the initial and the goal conditions of the threat profile, there is no feasible complex attack that the threat can implement.

$NO\_POSSIBLE\_COUNTERMEASURE$ denotes that the set of goal conditions is a subset of the initial conditions. In other words, the goal may be immediately achieved without applying any rule. There may be other complex attacks that achieve the goal but, even if all of those attacks are stopped, the goal state is still reachable.

In all the other cases, the value $ATTACK\_POSSIBLE$ will be associated with the threat profile.

In the first two cases, the countermeasure selector is bypassed and the minimal set of countermeasures is empty. The final output file specifies whether the empty set represents the lack of complex attacks or the lack of feasible countermeasures.

(3) Complex Attack Set
The variable *complexAttackSet* of type *Paths* is a minimal subset set. The type Paths is an implementation of a minimal subset set as discussed in Sect. 4.2.

5.4.2. The generation of a complex attack set

A high level description of the implementation of the complex attack set generator is shown in Fig. 5.4.1 where the meaning of most lines should be straightforward, some lines merit a few comments:

- *reachableConditionsGeneration* is the function that implements the algorithm in Fig. 4.1.1
- *complexAttackSetGeneration* is the function that implements the algorithm 4.2.16.
- $createModelInCPLEXLP$ is the procedure that generates the intermediate output file, which is the minimal cut set problem modeled as an ILP problem in the CPLEX LP format.
- $findCountermeasures$ is the function that interfaces with the countermeasure selector.
- $finalizeOutputFile$ is the procedure that finalizes the output file by appending the lines that describe the chosen countermeasures for each threat profile.

5.5. The Countermeasure Selector

The countermeasure selector is essentially an ILP solver. It takes as input the intermediate output file in CPLEX LP format and it generates a minimal set of countermeasures. It is implemented using the GLPK [**3**] package and it calculates the *optimal* solution, i.e., the set of countermeasures with the minimum cost. The output is a minimal set of countermeasures expressed as an array of boolean values. The length of the array equals to the number of possible countermeasures and an element is true if the corresponding countermeasure belongs to the minimal set.

This is an optional component and may be substituted by other ILP solvers. In particular, as we are interested in finding *a* solution and not necessarily the optimal one, an ILP solver that computes a heuristic solution efficiently may be a feasible alternative.

```
1  Parser parser := new Parser(inputfile)
2  Rules rules := parser.getRules()
3  Index reachabilityIndex := fillReachabilityIndex(rules)
4  foreach id in parser.getNames() {
5    Set reachableConditions := reachableConditionsGeneration
6                                          (rules,
7                                           parser.getInitCond(id))
8    Paths complexAttacksSet := complexAttackSetGeneration
9                                          (parser.getGoal(id),
10                                          parser.getInitCond(id),
11                                          reachableConditions,
12                                          reachabilityIndex)
13   createModelInCPLEXLP(outputfile, complexAttackSet)
14   if (parser.getInitCond(id) in parser.getGoal(id))
15     attackability.put(id, NO_POSSIBLE_COUNTERMEASURE
16   else if (complexAttackSet.isEmpty)
17     attackability.put(id, NO_ATTACK_POSSIBLE)
18   else attackability.put(id, ATTACK_POSSIBLE)
19 }
20 String result := findCountermeasures(attackability,
21                                      parser.getNames(),
22                                      parser.getNumOfRules())
23 finalizeOutputFile(result)
```

FIGURE 5.4.1. A high level description of the implementation of the complex attack set generator

The countermeasure selector also implements an algorithm that chooses a subset from the minimal set of countermeasures so that the total cost of countermeasures in this subset is within the budget as discussed in Sect. 4.3.2. The problem of choosing this subset is modeled as a linear knapsack problem as follows:

$max \sum_{i=1..N} x_i v_i$

$\sum_{i=1..N} x_i w_i \leq W$

$x_i \in \{0,1\}, \forall i \in \{1,..,N\}$

where

- N is the cardinality of the minimal set of countermeasures
- $x_i = 1$ if the corresponding countermeasure is in the subset; $x_i = 0$ otherwise

- v_i is the cost of the corresponding countermeasure
- w_i is the cost of the corresponding countermeasure
- W is the budget

The constraint guarantees that the total cost of countermeasures of the subset does not exceed the budget. The above chosen semantic meaning of v_i implies that we should spend as much as possible, as long as the expenditure is within the budget. In other words, we should avoid being skimpy on the selection of the subset. There are other possible choices of the semantic meaning of v_i:

- if v_i is defined as 1 for every i, then the preference is to select the largest subset possible;
- if v_i is defined as the probability of the application of the corresponding rule, then the preference is to stop the more likely complex attacks if no alternative countermeasure is adopted for the unselected countermeasures;
- if v_i is defined as the inverse of the viability of an *alternative* countermeasure that inactivates the corresponding rule(s), then the preference is to select countermeasures that are unlikely to have alternative solution(s).

However, some of these values are not easily quantifiable and, therefore, they are not used in the current implementation.

As the countermeasure costs and the budget are non-negative real numbers, they need to be converted into integers, which can be done by multiplying by 10^m, where m is the maximum number of decimal places among these numbers.

The subset will be empty if the budget is smaller than the cost of the least expensive countermeasure of the minimal set.

5.5.1. Example 1

This example shows the behavior of the subset selection process and it makes use of the same example as in Sect. 4.5. In particular, the subset is selected *a posteriori* to the selection of the minimal set of countermeasures. In other words, at first we select a set of countermeasures Σ such that:

- the adoption of these countermeasures stops all the complex attacks;

- ❖ the sum of the costs of these countermeasures is the *minimum* among those of alternative complete sets of countermeasures.

Then, we select a subset σ_m of Σ such that the sum of the attribute[1] v corresponding to the countermeasures in σ_m is the *maximum* among those of all the subsets whose sum of the costs of countermeasures is within the budget. Formally, we select σ_m such that $\rho_m = \underset{i=1..k}{max}\{\rho_i\}$ where

- ❖ $\rho_i = \underset{c_j \in \sigma_i}{\Sigma} v_j$ where v_j is the value of the attribute v corresponding to the countermeasure c_j, for $i = 1..k$
- ❖ $\{\sigma_1, ..., \sigma_k\}$ is the power set of Σ

The output file can be found in Appendix A.4. Only the output file is shown as the content of the input file is contained entirely in the output one.

The complex attack set generated is $\{\{1, 2, 5, 8, 12, 18\}, \{1, 2, 5, 10, 12, 15, 18\}, \{1, 2, 5, 9, 13, 15, 18\}, \{1, 2, 5, 10, 13, 15, 18\}\}$ and the minimal set of countermeasures is $\{12, 13\}$ given the costs in the input file. The total cost of countermeasures in this set is 10, which exceeds the budget of 8. A subset is selected with preference on maximizing the expenditure. Therefore, the countermeasure enumerated 13 , whose cost is 7, is chosen over that enumerated 12, whose cost is 3.

We observe that countermeasures enumerated 1, 2, 5 are common to all the elements of the complex attack set, which makes them ideal candidates as elements of the minimal set of countermeasures. However, as their costs are the symbolic value max, they are not chosen as long as alternative sets of countermeasures are available. The alternative complete sets of countermeasures are $\{9, 10\}$, $\{9, 15\}$, $\{12, 13\}$, $\{12, 15\}$, whose costs are respectively $3 + 8$, $3 + 7.7$, $3 + 7$, $3 + 7.7$. Therefore, the set $\{12, 13\}$ is chosen. After that, the selection of the subset takes place and the chosen subset is $\{13\}$.

[1] In this particular case, v_i is defined as the cost of the countermeasure i.

If these two operations were organized *not* in cascade, for example, if the subset S was selected such that the sum of the attribute v of the countermeasures in S is the minimum among that of other elements in U where U is union of all the power sets of complete set of countermeasures. Formally, $U = \{t : t \subseteq T \wedge T \in \Psi\}$ and Ψ is the set of all complete sets of countermeasures. Then, the subset $\{10\}$ would have been chosen as the subset. □

5.5.2. Example 2

This example shows the impact of choosing different semantic meanings of v_i. Suppose that

- the minimal set of countermeasures is $\{c_1, c_2, c_3, c_4, c_5\}$
- the cost of the countermeasures $c_1, ..., c_5$ are 7, 4.5, 3, 9, 1, respectively
- the budget is 17

Then the subset will be:

- $\{c_5, c_1, c_4\}$ if v_i is defined as the cost of c_i . In this case, the expenditure is maximized.
- $\{c_5, c_2, c_3, c_1\}$ if v_i is defined as 1, for all the countermeasures. In this case, a subset of the largest cardinality is preferred.
- $\{c_3, c_2, c_4\}$ if v_i is defined as the probability of the application of the rules corresponding to the countermeasures and that the probability are 0.5, 0.3, 0.4, 0.7, 0.1. In this case, the preference is to inactivate the more probable rules.
- $\{c_5, c_4, c_2\}$ if v_i is defined as the inverse of the viability of an alternative countermeasure relative to c_i and that the viability are $5, 4, 10, 2, 5$. In this case, the preference is to choose countermeasures that are unlikely to have an alternative.

It is up to the user to decide what to do with the leftover countermeasures. For example, an alternative countermeasure to the removal of a vulnerability may be to forbid the execution of the corresponding application. If v_i is defined as the

probability of the successful application of the corresponding rule, then the rules corresponding to the leftover countermeasures may have such a low probability that the residual risk is acceptable. □

Chapter 6

Evaluation

This section presents a set of tests that have been implemented to evaluate the developed toolkit. The test results are based upon analyses of a simulated network. All the model networks and their respective parameters are closely based upon Lippman *et al.* [**17**]. As the actual collection of input data is not part of this thesis, the only way to generate a large amount of input data is through a simulated network. Another reason in favor of considering a simulated network is that each aspect of the simulated network can be modified so as to investigate the scaling performance of this toolkit.

The experiments described in this chapter show the scalability of this tool. It is possible to analyze a complex network through a suitable choice of abstraction level and network model.

6.1. Network model I: Flat Network

A flat network contains H fully connected hosts with no firewall is shown in Fig. 6.1.1. Each host has 10 open ports, each of which permits access to a single remote-to-other vulnerability. $h\%$ of the hosts have the first remote-to-other vulnerability replaced with a remote-to-root vulnerability and are therefore compromisable. The goal of the attacker is to obtain root-level privileges on any of the compromisable hosts. The attacker can attack all the compromisable hosts, then these compromised

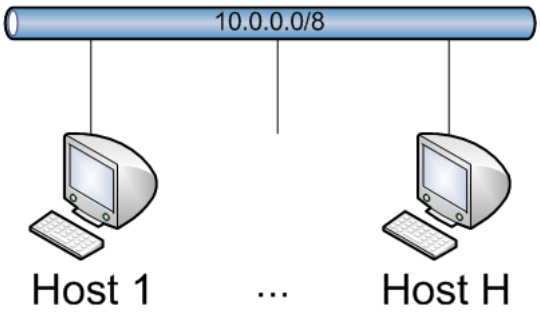


FIGURE 6.1.1. The configuration of a flat network

% Rootable	Total Hosts	T_1	T_2	T_3	Nodes	$\|R\|$	$\|\Omega\|$
5	125	21	5	5	7	42	5
5	250	23	7	6	13	156	6
5	500	24	13	17	26	650	17
5	1000	35	24	60	51	2550	60
5	2000	184	53	246	101	10100	246
5	4000	2515	151	1159	201	40200	1159

FIGURE 6.1.2. Test results of the simulation for a flat network as the number of hosts varies

hosts can in turn infect other compromisable hosts in the network. The parameters of interest of the network are H and h . The experiment on the flat model network uses between 125 and 4000 hosts, of which five percent are compromisable. Fig. 6.1.2 shows the experimental parameters and the data generated. Each row represents the data generated in a single run. The values of each column have the following meanings:

- % rootable: the percentage of hosts with remote-to-root vulnerability
- T_1: the time used to generate the set of reachable conditions, measured in milliseconds
- T_2: the time used to generate the complex attack set, measured in milliseconds
- T_3: the time used to generate the minimal set of countermeasures, measured in milliseconds
- nodes: the number of visited condition nodes in the condition graph
- $|R|$: the number of rules generated
- $|\Omega|$: the cardinality of the complex attack set

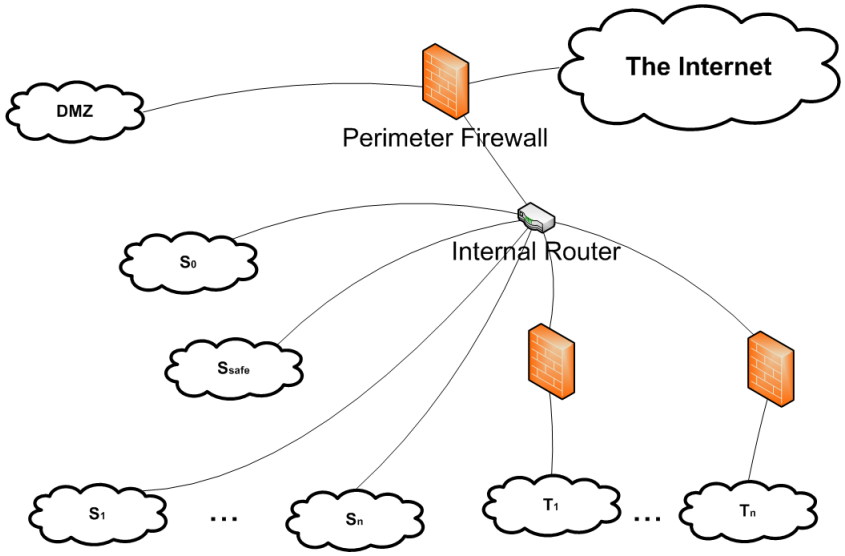


FIGURE 6.2.1. Enclave network block diagram

The set of reachable conditions is computed in linear time with respect to the number of rules. This is due to the fact that the actual complexity of the algorithm in Fig. 4.1.1 depends upon how many times the list of rules needs to be scanned. If we assign a value c_v to each condition c to represent the minimum number of rules to be applied to enable c, then the complexity is $O(|R| \cdot \underset{c \in C}{max}\{c_v\})$ where C is the set of all conditions. In this flat network, for every condition c , a larger number of hosts does not result in an increase in the minimum number of rules to be applied to enable c. Therefore, $\underset{c \in C}{max}\{c_v\}$ remains constant as H varies. The number of nodes generated shows that the pruning of the condition graph is very effective in this simple example.

In this network, the number of rules explodes as number of hosts grows. We will see in later sections that by collapsing similar hosts together [**12**], the number of rules grows linearly in the number of hosts.

6.2. NETWORK MODEL II: ENCLAVE NETWORK

The enclave model represents a typical network of a medium-sized corporate or a government site. The network configuration is shown in Fig. 6.2.1. The enclave network is separated from the Internet by a perimeter firewall with three interfaces:

Total Hosts	T_1	T_2	T_3	Nodes	|R|
267	45	11	60	138	2909
517	81	18	117	263	5784
1017	235	29	245	513	11534
2017	878	54	499	1013	23034
4017	5253	140	1021	2013	46034

FIGURE 6.2.2. Test results for an enclave network as the number of hosts varies

one on the Internet, one on the DMZ subset and one on an internal router. There are at least two subnets S_0 and S_{safe} connected to this router and up to 253 subnets of which up to 30 tenants. The characteristics of these subnets are as follows:

- DMZ: all the hosts in this subnet are compromisable via a remote-to-root vulnerability. The perimeter firewall is set up in a way such that
  - from the Internet, only one host in the DMZ is reachable. This host will then be used as a stepping stone [**25**] to compromise internal hosts,
  - there is only one host in S_0 that is reachable from all the hosts in the DMZ;
- S_0: all of the hosts in this subnet are compromisable;
- S_{safe}: this subnet contains up to 254 hosts. Each host is fully patched and without vulnerability;
- tenants: these are subnets separated from the rest of the internal network through additional firewalls. All the hosts on Tenant 1 are compromisable;
- other subnets: each of these subnets contains up to 254 hosts, 50% of which are compromisable.

The attacker is located outside the security perimeter. The goal of the attacker is to compromise as many hosts as possible. The experiment on the enclave network uses between 267 and 4017 hosts. Fig. 6.2.2 shows the experimental parameters and the data generated. Each row represents the data generated in a single run.

Degree of Explosion	Total Hosts	T_1	T_2	T_3	Nodes	|R|
1	1017	235	30	245	513	11534
4	1020	240	30	244	516	11552
9	1025	243	29	250	521	11622
16	1032	248	30	244	528	11804
25	1041	262	30	255	537	12182
36	1052	282	31	275	548	12864

FIGURE 6.3.1. Test results for an enclave network as the degree of a single-level firewall explosion varies

6.3. NETWORK MODEL II: ENCLAVE NETWORK, FIREWALL EXPLOSION

In the previous network there is only one potential path into the network. In this experiment, we are going to introduce additional vulnerable hosts into the subnet S_0 so that the attacker can proceed to the compromisation of the rest of the internal network from any compromised host in S_0. The results are shown in Fig. 6.3.1. The computation is not affected by the insertion of further hosts in S_0, because

- the *minimum* number of rules to be applied to enable any condition does not increase if additional hosts are inserted into S_0. Therefore, T_1 remains fairly constant.
- the complex attack set *covers* all the complex attacks. Since every complex attack a that compromises a host h in S_0 is covered, there is no need to consider any complex attack that uses h as stepping stone because if we prevent an attacker from controlling the stepping stone, subsequent complex attacks are automatically stopped as well. Therefore, T_2 remains fairly constant.

Fig. 6.3.2 shows the results of firewall explosions in series, where

- "degree of first explosion" indicates the number of hosts placed in the DMZ
- "degree of second explosion" indicates the number of hosts placed in S_0

To show in more details the problem of firewall explosion, we change the goal of the attacker to the compromisation of hosts not belonging to the DMZ nor to S_0. The configuration of the experiment is the same as the one used to generate the

Degree of First Explosion	Degree of Second Explosion	Total Hosts	T_1	T_2	T_3	Nodes	|R|
1	1	1017	235	29	245	513	11534
2	2	1019	272	28	265	515	12561
3	3	1021	329	32	298	517	13592
4	4	1023	358	32	331	519	14627
5	5	1025	411	33	337	521	15666
6	6	1027	464	32	360	523	16709

FIGURE 6.3.2. Test results for an enclave network as the degrees of two firewall explosions varies

Degree of First Explosion	Degree of Second Explosion	Total Hosts	T_1	T_2	T_3	Nodes	|R|
1	1	1017	233	1435	428	1531	11532
2	2	1019	271	20310	795	6631	12557
3	3	1021	314	947183	3214	46411	13586

FIGURE 6.3.3. Test results for an enclave network as the degrees of two firewall explosions varies, with a modified goal

Degree of First Explosion	Degree of Second Explosion	T_1	T_2	T_3	Nodes	|R|
n	n	22	2	6	7	10

FIGURE 6.4.1. Test results for an enclave network as the degrees of two firewall explosions varies, with hosts collapsed

test results in Fig. 6.3.1. The results are shown in Fig. 6.3.3. The phenomenon of firewall explosion happens in real networks, thereby making impractical the analysis of even a moderately sized network with a series of filtering devices. This is very disturbing since these networks results when applying important security strategies such as “defense in depth”.

6.4. Network model II: Enclave Network, hosts collapsing

To workaround the problem of firewall explosion, we adopt the host collapsing model in Lippmann *et al.* [**17**]. Two hosts are collapsed into a single one if they can be

compromised at the same level. This often happens in a subnet of real networks where all the hosts are configured in a fairly similar way. When one of the hosts in such subnet is compromised, the rest in the same subnet can be compromised as well. The results are shown in Fig. 6.4.1. Only a single run is executed as the degree of explosion is irrelevant to the results, be it degree 1 or degree 100, the whole subnet is collapsed into a single host.

6.5. Network model II: Enclave Network, better hosts collapsing

The adoption of a collapsing strategy strongly improves the scalability but it results in a loss of all the details of the inner behavior of a subnet. Therefore, we propose a different collapsing strategy. For each subnet s we define a variable v_s and insert a rule $r : h_s \rightarrow v_s$ for each host h_s in the subnet. The cost of the countermeasure for this rule is the symbolic value max. By adopting this strategy, the analysis of the network remains tractable. The test results are shown in Fig. 6.5.1. The configuration of the network is the same as that used in Sect. 6.4.

Example 6.5.1.
Suppose that

- there are two subnets s_1 and s_2 with hosts $h^1_{s_1}, h^2_{s_1}, ..., h^{n_{s_1}}_{s_1}$ and $h^1_{s_2}, h^2_{s_2}, ..., h^{n_{s_2}}_{s_2}$, respectively
- each host has one remote-to-root vulnerability.
- every host in s_1 can compromise every host in s_2

If the collapsing model is not adopted, there are $n_{s_1} \cdot n_{s_2}$ rules: $h^i_{s_1} \rightarrow h^j_{s_2}$ for $i = 1...n_{s_1}$, $j = 1..n_{s_2}$. The adoption of the collapsing model in Lippmann *et al.* [**17**] results in a single rule that says that subnet s_2 is reachable from subnet s_1. The collapsing model we proposed defines $n_{s_1} + n_{s_2}$ rules: $h^i_{s_1} \rightarrow v_{s_1}$ for $i = 1..n_{s_1}$ and $v_{s_1} \rightarrow h^j_{s_2}$ for $j = 1..n_{s_2}$.

Degree of First Explosion	Degree of Second Explosion	Total Hosts	T_1	T_2	T_3	Nodes	|R|
1	1	1017	47	1840	172	4591	2558
2	2	1019	47	1924	171	6631	2564
3	3	1021	31	2090	172	8671	2570
4	4	1023	31	2309	172	10711	2576
5	5	1025	31	2356	171	12751	2582
6	6	1027	31	2558	173	14791	2588

FIGURE 6.5.1. Test results for an enclave network as the degrees of two firewall explosions varies, with an alternative collapsing strategy

Appendices

Appendix A

Grammar of the input file language

The grammar of the input language[1] is defined by the following productions. The notational conventions are as follows:

- ❖ choices are separated by vertical bars: |
- ❖ optional parts are enclosed in square brackets: [...]
- ❖ A Kleene star * indicates zero or more repetitions of the immediately preceding fragment
- ❖ literals are enclosed in single quotes: ' ... '
- ❖ lowercase names refer to tokens representing keywords
- ❖ uppercase names refer to grammar productions from this list.

```
Spec ::= Lines
Lines ::= Line
        | Line Lines
Line ::= Variable
       | Countermeasure
Variable ::= '#' default '=' Real
           | '#' goal '(' Name ')' '=' '{' Conditions '}'
           | '#' init '(' Name ')' '=' '{' ZeroOrMoreConditions '}'
           | '#' budget '(' Name ')' '=' '{' Real '}'
Countermeasure ::= ['(' Cost ')'] Rules
```

[1] with the comments stripped

```
Cost ::= max
       | Real
Rules ::= Rule
        | Rule ';' Rules
Rule ::= ZeroOrMoreConditions '->' Conditions
ZeroOrMoreConditions ::= Conditions
                       |
Conditions ::= Condition
             | Condition ',' Conditions
Condition ::= Num
            | Num Digit*
Num ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Digit ::= '0' | Num
Real ::= Digit Digit* ['.' Digit*]
Name ::= Alphabet
       | Alphabet Alphabet*
Alphabet ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
           | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p'
           | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x'
           | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
           | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
           | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V'
           | 'W' | 'X' | 'Y' | 'Z' | '_'
```

A.1. An example input file

```
// ------- SECTION: VARIABLES ------- //

// default countermeasure cost
# default = 1.0

// goal conditions, there are two threat profiles,
// identified as "insider" and "outsider"
# goal(insider) = {5}
# goal(outsider) = {8}

// initial conditions
# init(insider) = {1,3}
# init(outsider) = {5}

// ------- SECTION: RULES ------ //

// the following rules have countermeasures with the default cost
1 -> 2,3  // precondition: 1; postconditions: 2 and 3
3 -> 2,4  // precondition: 3; postconditions: 2 and 4
```

```
6 -> 7 // precondition: 6; postcondition: 7

// a rule whose countermeasure cost is 5.5
(5.5) 2 -> 1

// a single countermeasure that stops two rules
(7) 1,4 -> 5; 7 -> 5

// there is no countermeasure for the following rule
(max) 7 -> 8
```

A.2. Grammar of the output file language

The output file language is just a slight modification of the input file language as discussed in Sect. 5.2.1

```
Spec ::= Lines
Lines ::= Line
        | Line Lines
Line ::= Variable
       | Countermeasure
Countermeasure ::= ['(' Cost ')'] Rules
Variable ::= '#' default '=' Real
           | '#' goal '(' Name ')' '=' '{' Conditions '}'
           | '#' init '(' Name ')' '=' '{' Conditions '}'
           | '#' countermeasure '(' Name ')' '=' '{' Countermeasures '}'
           | '#' budget '(' Name ')' '=' '{' Real '}'
           | '#' countermeasuresWithinBudget '(' Name ')' '=' '{' Countermeasures '}'
Countermeasures ::= Enumeration '.' ['(' Cost ')'] Rules
Cost ::= max
       | Real
Rules ::=  Rule
        | Rule ';' Rules
Enumeration ::= Num
              | Num Digit*
Rule ::= ZeroOrMoreConditions '->' Conditions
ZeroOrMoreConditions ::= Conditions
                       |
Conditions ::= Condition
             | Condition ',' Conditions
Condition ::= Num
            | Num Digit*
Num ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
Digit ::= '0' | Num
Real ::= Digit Digit* ['.' Digit*]
Name ::= Alphabet
       | Alphabet Alphabet*
```

```
Alphabet ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
           | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p'
           | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x'
           | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
           | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
           | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V'
           | 'W' | 'X' | 'Y' | 'Z' | '_'
```

A.3. An example output file

The example input file in A.1 will generate the following output file:

```
// ------- SECTION: VARIABLES ------- //
// default countermeasure cost
# default = 1.0

// goal conditions, there are two threat profiles,
// identified as "insider" and "outsider"
# goal(insider) = {5}
# goal(outsider) = {8}

// initial conditions
# init(insider) = {1,3}
# init(outsider) = {5}

// ------- SECTION: RULES ------- //

// the following rules have countermeasures with the default cost
1.       1 -> 2,3
2.      3 -> 2,4
3.      6 -> 7

// a rule whose countermeasure cost is 5.5
4.   (5.5) 2 -> 1

// a single countermeasure that stops two rules
5.       (7.0) 1,4 -> 5;7 -> 5

// there is no countermeasure for the following rule
6. (max) 7 -> 8

//*************** RESULT *****************//

# countermeasure(outsider) = {} // no complex attack possible
# countermeasure(insider) = {2}
```

A.4. The output file for the example in Sect.5.5.1

```
// default countermeasure cost
# default = 1.0

// goal condition
# goal(insider) = {6}

// initial conditions
# init(insider) = {14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32, \
33,34,35,36,37,38,39,40,41,42,43,44,8,1,2,3,4,5}

// budget
# budget(insider) = 8

// dependencies: root-level privileges implies user-level privileges
1. (max) 8 -> 11
2. (max) 7 -> 9
3. (max) 6 -> 10
4. (max) 53 -> 12

// IIS buffer overflow
5. (max) 11, 1, 27 -> 7
6. (5.5) 9, 1, 28 -> 7
7. (5.5) 12, 1, 31 -> 7
8. (5.5) 10, 1, 34 -> 7

// squid port scan
9. (3) 30, 9 , 2 -> 13
10. (8) 33, 12, 2 -> 13
11. (2.9) 36, 10, 2 -> 13

// LICQ remote-to-user
12. (3) 40, 9, 13, 3 -> 10
13. (7) 42, 12, 13, 3 -> 10
14. (3) 44, 10, 13, 3 -> 10

// client scripting
15. (7.7) 31, 9, 4 -> 12
16. (4) 32, 12, 4 -> 12
17. (4) 33, 10, 4 -> 12

// local setuid buffer overflow
18. (max) 10, 5 -> 6

//*************** RESULT *****************//
```

```
# countermeasure(insider) = {12, 13}

// Budget exceeded. Total cost of countermeasures for insider = 10.0
# countermeasuresWithinBudget(insider) = {13}
```

Appendix B

Source Code

The source code of the toolkit, structured into three components.

B.1. The Input Parser

```
package input;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Hashtable;
import java.util.Set;
import java.util.Vector;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import utils.Log;
import utils.Misc;

import main.Rules;

/**
 * Input parser
 * The only interface between the input file and the other components of the
 * toolkit. Should the input file format be changed, modify this file
```

```
 * accordingly, providing the same getter methods
 */
public class Parser{
    private double defaultCost = 1;
    private Vector<Double> countermeasureCost = new Vector<Double>();
    private double sumOfCountermeasureCosts = 0;
    private Rules rules;
    private int nextID = 1; // the next rule ID to be assigned, starting from 1

    // Total number of countermeasures. Each rule line in the input file is
    // counted as one countermeasure. A single countermeasure corresponds to
    // multiple rules when:
    // 1. A rule has more than one postcondition
    // OR
    // 2. Multiple rules are listed on the same line
    private int numOfRules = 0;

    // the following hashtable maps the identifier of a threat profile to
    // the set of goals, the set of initial conditions and the budget respectively
    private Hashtable<String, int[]> goal = new Hashtable<String, int[]>();
    private Hashtable<String, int[]> condInit = new Hashtable<String, int[]>();
    private Hashtable<String, Double> budget = new Hashtable<String, Double>();

    /**
     * REQUIRES: file exists and is readable
     * @param file the input file
     * @throws IllegalArgumentException if the precondition is violated
     * @throws IllegalGoalException if failed to parse a valid goal from file
     */
    public Parser(String file, String outputFileName)
        throws IllegalGoalException, IllegalInitialConditionException {
        rules = new Rules();

        // preparing input file
        FileInputStream fstream = null;
        try { fstream = new FileInputStream(file);
        } catch (FileNotFoundException e) {
            throw new IllegalArgumentException("The file " + file +
                    " is not found.");}
        DataInputStream in = new DataInputStream(fstream);
        BufferedReader br = new BufferedReader(new InputStreamReader(in));

        // preparing output file
        FileWriter foutstream = null;
        try { foutstream = new FileWriter(outputFileName);
        } catch (IOException e) {
```

```
    throw new IllegalArgumentException("the file " + outputFileName +
            " cannot be opened for writing");}
BufferedWriter out = new BufferedWriter(foutstream);

String line;
try {
  while ((line = br.readLine()) != null)   {
    String lineCopy = line; // copy the line to the output file
    String content = line.split("//")[0].trim(); // strip comments
    if (!content.equals("")) {
      if (content.matches
  ("^\\s*#\\s*[dD][eE][fF][aA][uU][lL][tT]\\s*=\\s*.*"))
            parseDefaultCost(content);

      else if (content.matches
  ("^\\s*#\\s*[gG][oO][aA][lL]\\s*\\(\\s*\\w*\\s*\\)\\s*=\\s*.*"))
            parseGoal(content);

      else if (content.matches
  ("^\\s*#\\s*[iI][nN][iI][tI]\\s*\\(\\s*\\w*\\s*\\)\\s*=\\s*.*"))
            parseInitCond(content);

      else if (content.matches
  ("^\\s*#\\s*[bB][uU][dD][gG][eE][tT]\\s*\\(\\s*\\w*\\s*\\)\\s*=\\s*.*"))
            parseBudget(content);

      else if (content.matches("^\\s*#.*")) {
          Log.errorLog("Unknown variable defined in the line: " + line);
          lineCopy = ""; } // do not copy unknown lines to the output file
      else
          lineCopy = parseRule(content);
    }
    try { out.append(lineCopy + "\n");
    } catch (IOException e) {
        Log.errorLog("error in outputing the line: " + nextID + ".\t" +
                content + "\n" + " to the output file"); }
  } // end while

  // closing files
  in.close();
  out.close();
} catch (IOException e) {}
finally {
   if (in != null)
       try {in.close();}
       catch (IOException e) {e.printStackTrace();}
```

```
        if (out != null)
            try {out.close();}
            catch (IOException e) {e.printStackTrace();}
    }
}

// Parse the budget line (case insentive, white space ignored)
// #budget(<Name>) = <PositiveRealNumber>
private void parseBudget(String content) {
    Matcher m = Pattern.compile
          ("^\\s*#\\s*[bB][uU][dD][gG][eE][tT]\\s*\\(\\s*(\\w*)\\s*\\)\\s*" +
           "=\\s*(\\d+\\.?\\d*)\\s*").matcher(content);
    // group(1) is the threat profile identifier
    // group(2) is the budget
    if (m.matches() && m.group(1) != null && m.group(2) != null)
        try {
            double budgetValue = Double.parseDouble(m.group(2));
            Double old = budget.put(m.group(1), new Double(budgetValue));
            if (old != null)
                Log.errorLog("The budget of " + m.group(1) +
                        " already exists. The previous value " +
                        old + " will be replaced.\n");
            else  Log.log("The budget of " + m.group(1) + " is "+budgetValue);
        } catch (Exception e) {
            Log.error("invalid line: "+content+"\nInvalid budget, no budget"
                    + " will be defined for " + m.group(1));
        }
    else
        Log.errorLog("Error in parsing budget from the line: " + content);
}

// Parse the rule line (case insentive, white space ignored)
// [\(max|<NonNegativeRealNumber>\)] <Rule> (;<Rule>)*
// Rule ::= [<Int>](,<Int>)* -> <Int>(,<Int>)*
// [] = optional, () = grouping, <Int> = positive integer > 0
private String parseRule(String content) {
    String result = "";
    Double cost = defaultCost;
    boolean defaultCost = true;
    String rulePattern = "\\s*(?:\\d+\\s*)?+(?:,\\s*\\d+\\s*)*\\s*->" +
            "\\s*(?:\\d+\\s*,\\s*)*\\d+\\s*";
    Matcher m = Pattern.compile
        ("(?:\\(\\s*((?:[mM][aA][xX])|\\d+\\.?\\d*)\\s*\\))??((?:" +
        rulePattern + ";\\s*)*" + rulePattern + ")").matcher(content);
    if (m.matches() && m.group(2) != null) {
        // parse countermeasure cost
```

```
try {
    if (m.group(1) != null)
        if (m.group(1).equalsIgnoreCase("max"))
            cost = null;
        else { cost = Double.parseDouble(m.group(1));
            defaultCost = false;
        }
} catch (Exception e) {
    Log.error("Error in parsing countermeasure cost for the rule "
            + content + ".\nThe default countermeasure cost " +
          defaultCost + " will be used.");}

// parse rules
result += nextID + ".\t" +
    (cost == null? "(max) " : (defaultCost? "" : "(" + cost + ") "));
String[] contents = m.group(2).split(";");
boolean ok = false;
for (String c : contents) {
  content = c.trim();
  int[] preconditions = null;
  int[] postconditions = null;
  try {
    String[] preconditionString =
        content.split("->")[0].trim().split(",");
    String[] postconditionString =
        content.split("->")[1].trim().split(",");
    preconditions = new int["".equals(preconditionString[0]) ? 0 :
                    preconditionString.length];
    postconditions = new int[postconditionString.length];
    ok = true;
    for (int i = 0; i < preconditions.length; i++) {
        preconditions[i] =
            Integer.parseInt(preconditionString[i].trim());
        if (preconditions[i] <= 0) throw new Exception(); }
    for (int i = 0; i < postconditions.length; i++) {
        postconditions[i] =
            Integer.parseInt(postconditionString[i].trim());
        if (postconditions[i] <= 0)
            throw new Exception(); }
  } catch (Exception e) { ok = false;   }
  if (ok) {
    for (int i = 0; i < postconditions.length; i++)
      rules.insert(new Rule(nextID,preconditions,postconditions[i]));
    result += content + ";";
  }
  else Log.error("Error in parsing the rule: " + content +
```

```
                    ". This rule will not be considered");
            } // end for
            if (ok) { // the whole line is correct
              countermeasureCost.add(cost);
              if (cost != null) sumOfCountermeasureCosts += cost;
              numOfRules++;
              nextID++;
            }
            return result.substring(0,result.length()-1) +"\n";
        }
        else {
            Log.errorLog("unrecognized line: " + content);
            return result;}
    }

    /*
     * Parse profile specific conditions which are in the form
     * #<profileType>(<profileName>) = {<Conditions>}
     * The parsed conditions will be mapped to the keys of the corr. hashtable
     * @param profiles the hashtable where the parsed info is saved
     * @param profileType the type of profile, in the current implemention it can
     *      be either "goal" or "init" which stand respectively for goal
     *      profiles and initial condition profiles
     * @param content the line to be parsed
     * @throws IllegalVariableException if the line is ill-formatted
     */
    private void parseProfileSpecificConditions(Hashtable<String, int[]> profiles,
            String profileType, String content) throws IllegalVariableException {
        assert(profiles != null && profileType != null && content != null);
        String varName = "";
        for (char c : profileType.toCharArray())
            varName += "[" + String.valueOf(c).toLowerCase() +
            String.valueOf(c).toUpperCase() + "]";
        Matcher m = Pattern.compile("#\\s*"+varName+"\\s*\\(\\s*(\\w*)\\s*\\)" +
            "\\s*=\\s*\\{\\s*((?:\\d+,\\s*)*\\d+)*\\s*\\}\\s*").matcher(content);

        // initialization for the empty initial condition case
        int[] conditions = new int[0];

        String name = "";
        // there is a bug in the regex implementation of Java, it gives stack
        // overflow error when the input string is too long
        // ref http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5050507
        // ref http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4675952
        if (content.length() < 2700) {
            if (m.matches()) {
```

```
        name = m.group(1).trim();
        if (!(m.group(2) == null))  { // non empty set of init conds
            String[] conds = m.group(2).split(",");
            conditions = new int[conds.length];
            for (int i = 0; i < conds.length; i++) {
                try {
                    int val = Integer.parseInt(conds[i].trim());
                    if (val <= 0) throw new Exception();
                    conditions[i] = val;
                } catch (Exception e) {
                    throw new IllegalVariableException
                    ("Error in parsing the " + profileType +
                            " conditions from the line " + content);}}}
            int[] old = profiles.put(name, conditions);
            if (old != null)
                Log.errorLog("The name " +  name +" already exists for "+
                        "another profile. The previous value :" +
                        old != null ? Misc.intArrayToString(old) : "{}" +
                                " will be replaced.\n");
            else  Log.log("The " + profileType + " conditions for " +
                    name + " are: " + Misc.intArrayToString(conditions));
    } // end if m.matches()
    else Log.errorLog("The line: \n" + content + "\nis supposed to " +
            "describe the "+ profileType +" conditions profile " +
                    "but for some reasons it is not parsed");
} // end if content.length() < 2700
else { // length >= 2700
    String[] tmp = content.split("=");
    boolean validInput = (tmp.length == 2);
    if (validInput) {//# init(insider)
        Matcher m1 = Pattern.compile("#\\s*" + varName +
                "\\s*\\(\\s*(\\w*)\\s*\\)\\s*").matcher(tmp[0]);
        validInput = m1.matches();
        name = m1.group(1).trim();
        String[] conditionsStrings =tmp[1].replace('{', ' ').
                                    replace('}',' ').split(",");
        conditions = new int[conditionsStrings.length];
        int i = 0;
        while (validInput && i < conditionsStrings.length) {
            try {
                int val = Integer.parseInt(conditionsStrings[i].trim());
                if (val <= 0)
                    throw new Exception();
                conditions[i] = val;
            } catch (Exception e) {
                validInput = false; }
```

```
                i++;
            } // end while
            if (validInput) {
                int[] old = profiles.put(name, conditions);
                if (old != null)
                    Log.errorLog("The name " + name + " already exists for "+
                            "another profile. The previous value :" +
                            old != null ? Misc.intArrayToString(old) : "{}"
                                + " will be replaced.\n");
                else  Log.log("The " + profileType + " conditions for " +
                        name + " are: " + Misc.intArrayToString(conditions));
                           }
            else Log.errorLog("The line: \n" + content
                    + "\nis supposed to describe the "+ profileType +
            " conditions profile but for some reasons it is not parsed"); }
    } // end else (length >= 2700)
}

/**
 * Parse the initial conditions line (case insentive, white space ignored)
 * #init(<Name>) = {[<PositiveRealNumber>] (, <PositiveRealNumber>)*}
 * @throws IllegalInitialConditionsExcepiton if failed to parse the initial
 *                                          conditions
 */
private void parseInitCond(String content)
throws IllegalInitialConditionException {
    try {
        parseProfileSpecificConditions(condInit,"init", content);
    } catch (IllegalVariableException e) {
        throw new IllegalInitialConditionException(e.toString());
    }
}

/**
 * Parse the goal conditions line (case insentive, white space ignored)
 * #goal(<Name>) = {<PositiveRealNumber> (, <PositiveRealNumber>)*}
 * @throws IllegalGoalException if fail to parse a valid goal from file
 */
private void parseGoal(String content) throws IllegalGoalException {
    try {
        parseProfileSpecificConditions(goal,"goal", content);
    } catch (IllegalVariableException e) {
        throw new IllegalGoalException(e.toString());
    }
}
```

```
// Parse the default cost line (case insentive, white space ignored)
// #default = <PositiveRealNumber>
private void parseDefaultCost(String content) {
    Matcher m = Pattern.compile("#\\s*[dD][eE][fF][aA][uU][lL][tT]\\s*=" +
            "\\s*(\\d+\\.?\\d*)\\s*").matcher(content);
    if (m.matches() && m.group(1) != null)
        try {
            defaultCost = Double.parseDouble(m.group(1));
            Log.log("The default countermeasure cost is " + defaultCost);
        } catch (Exception e) {
            Log.errorLog("invalid line: "+ content
                    + "\nThe default countermeasure cost 1 will be used");
        }
    else
        Log.errorLog("Error in parsing the default countermeasure cost " +
                "from the line: " + content
                + "\nThe default countermeasure cost 1 will be used");
}

/************************ Getter Methods ************************/

/**
 * Get the list of goals given the identifier of a threat profile
 * @param name identifier of a threat profile
 * @throws IllegalGoalException if there is no goal associated with name
 * @return the list of goals
 */
public int[] getGoal(String name) throws IllegalGoalException {
    if (goal.containsKey(name)) {
        int[] result = goal.get(name);
        if (result != null)
            return result;
        else
            throw new IllegalGoalException("There is no goal specified for" +
                    " the profile " + name);
    }
    else throw new IllegalGoalException("The goal profile " + name +
            " does not exist.");
}

/**
 * Get the sum of costs of all the countermeasures whose costs are not "max"
 * For defining "max" as sum+1
 * @return the sum
 */
public double getSumOfCountermeasureCosts() {
```

```
    return sumOfCountermeasureCosts;}

/**
 * @param name identifier of a threat profile
 * @return the initial conditions associated with name. If the set of initial
 *         conditions is empty, then an array of length zero (new int[0])
 *         will be returned.
 * @throws IllegalArgumentException if name does not correspond to any
 *         threat profile
 */
public int[] getInitCond(String name) {
    // we cannot check if condInit.get(name) is null as an initial conditions
    // profile may have zero conditions, which is valid. Instead it is an
    // exception if the specified name does not correspond to any name
    if (condInit.containsKey(name))
        return condInit.get(name);
    else throw new IllegalArgumentException("The name " + name +
                " does not correspond to any inital condition profile.");
}

/**
 * Get the total number of rule lines (i.e. total no. of countermeasures)
 * @return the total number of rule lines
 */
public int getNumOfRules() {
    return numOfRules;}

/**
 * Get names of all the threat profiles
 * @return the set of threat profile names
 */
public Set<String> getNames() {
    return condInit.keySet();   }

/**
 * Get an array of countermeasure costs. The countermeasure cost of a rule
 * with ID i is the (i-1)th element of the array.
 * e.g. The countermeasure cost of a rule with id 5 is given by
 *      getCountermeasureCost()[4]
 * @return an array of countermeasure costs
 */
public Double[] getCountermeasureCost() {
    Double[] result = new Double[numOfRules];
    int i = 0;
    for (Double v : countermeasureCost) {
        if (v == null) result[i] = sumOfCountermeasureCosts+1;
```

```
            else result[i] = v;
            i++;
        }
        return result;
    }

    /**
     * Get the rules
     * @return the rules
     */
    public Rules getRules() {
        return rules;
    }

    /**
     * Get the countermeasures budget relative to a threat profile
     * @param name identifier of a threat profile
     * @return the countermeasures budget, null if there is no budget
     *         (budget = infinite)
     */
    public Double getBudget(String name) {
        if (budget.containsKey(name))
            return budget.get(name);
        else return null;
    }
}
```

```
package input;

/***
 * Implementation of the abstract representation of a rule
 * An example rule:
 *  20. 2,3,4 -> 5
 * where 20 is the id, 2,3,4 are the preconditions and 5 is the postcondition
 */
public class Rule {
    // id, preconditions, postcondition must be > 0

    // the id of the rule, the id is NOT unique
    // in particular, when a rule with multiple postconditions is split into
    // multiple subrules, all these subrules will have the same id
    private int id;

    private int[] preconditions;
    private int postcondition;
```

```
/** Constructor
 * REQUIRES: id > 0, postcondition > 0, [c > 0 | c <- preconditions]
 * preconditions can be null, in that case, the rule can always be activated
 * @param id the ID of the rule
 * @param preconditions preconditions of the rule
 * @param postcondition postcondition of the rule
 * @throws IllegalArgumentException if the precondition is violated
 */
public Rule(int id, int[] preconditions, int postcondition) {
    if (id <= 0 || postcondition <= 0)
        throw new IllegalArgumentException("invalid id or postcondition");
    if (preconditions != null)
        for (int c : preconditions)
            if (c <= 0)
                throw new IllegalArgumentException
                ("invalid precondition of the rule with id " + id);
    this.id = id;
    this.preconditions = preconditions;
    this.postcondition = postcondition;
}

/**
 * two rules are equal when they have the same id
 */
public boolean equals(Object o) {
    if (o instanceof Rule)
        return this.id == ((Rule)o).id;
    return false;
}

public String toString() {
    String output = id + ": ";
    if (preconditions != null)
        for (int c : preconditions)
            output += c + ", ";
    output += "$-> " + postcondition; // avoid the last comma
    return output.replaceFirst(", \\$", " ").replace('$',' ');
}

public int getID() {
    return id; }

public int[] getPreconditions() {
  return preconditions; }
```

```
    public int getPostcondition() {
        return postcondition; }
}
```

B.2. The Complex Attack Set Generator

```
package main;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Hashtable;
import java.util.TreeSet;

import riskMitigation.GLPKerrorException;
import riskMitigation.Knapsack;
import riskMitigation.NoSolutionException;
import riskMitigation.Solve;
import riskMitigation.TimeoutException;

import utils.Log;
import utils.Set;
import input.IllegalGoalException;
import input.IllegalInitialConditionException;
import input.Parser;
import input.Rule;

/**
 * Implementation of the Complex Attack Set Generator component
 */
public class Main {
    private final static Integer NO_POSSIBLE_COUNTERMEASURE = 1;
    private final static Integer NO_ATTACK_POSSIBLE = 2;
    private final static Integer ATTACK_POSSIBLE = 0;

    // (for statistics only) count the number of condition nodes generated
    private static int counter = 0;

    /**
     * main entry point of this toolkit
     * @throws FileFormatException if there is a fatal error in parsing file
     */
    public static void main(String[] args) throws FileFormatException{
        // initialize the parser
        Parser p;
```

```
String inputFile = "regole.txt"; // input file name
String outputFileName = "regole(output).txt"; // output file name
try {
    p = new Parser(inputFile, outputFileName);
} catch (IllegalGoalException e) {
    throw new FileFormatException("illegal file format: Illegal goal");
} catch (IllegalInitialConditionException e) {
    throw new FileFormatException
        ("illegal file format: Illegal initial conditions"); }

Index reachabilityIndex = fillReachabilityIndex(p.getRules());
Hashtable<String,Integer> attackability = new Hashtable<String,Integer>();

for (String name : p.getNames()) {
    //long startTime = System.currentTimeMillis();
    Set reachableConditions =
        reachableConditionsGeneration(p.getRules(), p.getInitCond(name));
    //long stopTime = System.currentTimeMillis();
    //long runTime = stopTime - startTime;
    //System.out.println("reachable set generation run time: " + runTime);

    Paths complexAttacksSet;
    try {
        //startTime = System.currentTimeMillis();
        complexAttacksSet = complexAttackSetGeneration(
                reachableConditions,
                p.getGoal(name),
                new Set(p.getInitCond(name)),
                reachabilityIndex);
        //stopTime = System.currentTimeMillis();
        //runTime = stopTime - startTime;
        //System.out.println("complex attack set generation run time: "
        //  + runTime);

        Log.log("Statistics:\nTotal number of nodes generated = " +
                counter + "\n");

        // intermediate output file name
        String modelFile = "model(" + name + ").lp";

        createModelInCPLEXLP(modelFile,
                complexAttacksSet,
                p.getCountermeasureCost(),
                p.getNumOfRules(),
                p.getSumOfCountermeasureCosts());
```

```
            if (new Set(p.getInitCond(name)).contains(p.getGoal(name)))
                attackability.put(name, NO_POSSIBLE_COUNTERMEASURE);
            else if (complexAttacksSet.isEmpty())
                attackability.put(name, NO_ATTACK_POSSIBLE);
            else attackability.put(name, ATTACK_POSSIBLE);
        } catch (IllegalGoalException e1) {
            throw new FileFormatException("missing goal in the input file " +
                    "for the threat profile: " + name);
        }
    }

    //long startTime = System.currentTimeMillis();
    String result = findCountermeasures(
            attackability,
            p.getNames(),
            p.getNumOfRules(),
            p.getCountermeasureCost(),
            p);
    //long stopTime = System.currentTimeMillis();
    //long runTime = stopTime - startTime;
    //System.out.println("countermeasure generation run time: " + runTime);
    //System.out.println("number of rules: " + p.getNumOfRules());
    finalizeOutputFile(outputFileName, result);
}

/**
 * compute the set conditions reachable from the initial state
 * @param rules the set of rules
 * @param initConds the set of initial conditions
 * @return the set of reachable conditions
 */
private static Set reachableConditionsGeneration(Rules rules,int[] initConds){
    // init
    boolean modified = true;
    Set conditions = (initConds == null) ? new Set() : new Set(initConds);
    Queue q = new Queue();
    for (Rule r : rules)
        q.add(r);
    // temporary holder of the lists of rules during one pass of the rules
    Queue tmp = new Queue();
    Set newConditions = new Set();

    // This differs from the pseudocode as the list is modified during
    // iteration. A temporary list is used to simplify the implementation.
    // There is a great difference in efficiency of actually deleting items
    // from the list than simply marking elements as deleted.
```

```
    while (!q.isEmpty()) {
        Rule element = q.remove();
        if (conditions.contains(element.getPreconditions())) {
            modified = true;
            newConditions.addElement(element.getPostcondition());
        }
        else tmp.add(element);

        // when one pass of the rules is completed
        if (q.isEmpty() && modified) {
            modified = false;
            conditions.union(newConditions);
            newConditions = new Set();
            q = tmp;
            tmp = new Queue();
        }
    }
    return conditions;
}

/**
 * compute the Complex Attack Set
 * @param reachableConditions set of conditions reachable from init state
 * @param goals set of goal conditions
 * @param initConditions set of init conditions
 * @param reachabilityIndex
 * @return the Complex Attack Set
 */
private static Paths complexAttackSetGeneration(Set reachableConditions,
        int[] goals, Set initConditions, Index reachabilityIndex) {
    if (goals == null || goals.length == 0)
        throw new IllegalArgumentException("the goal cannot be empty");
    Paths result = new Paths();
    if (!initConditions.contains(goals)&&reachableConditions.contains(goals)){
        goals = Set.exclude(initConditions, goals); // G = G \ Init
        for (int i = 0; i < goals.length; i++) {
            Set included = new Set(goals);
            included.exclude(goals[i]); // I = G \ {g}

            result = Paths.product(result, findAttackPaths(
                    goals[i],
                    new Set(new int[]{goals[i]}),
                    included,
                    initConditions,
                    reachableConditions,
                    reachabilityIndex,
```

```
                    true)); }}
    return result;
}

// the auxiliary function f, each call to it creates a condition node
// @param isGoal true if c is one of the goal condition AND f is called
//               directly from findAttackPaths; false otherwise
// @param c the condition corresponding to the condition node being generated
private static Paths findAttackPaths(int c, Set excludedCondition,
        Set includedCondition, Set initConditions,
        Set reachableConditions, Index reachabilityIndex, boolean isGoal){
    counter++; // count no. of condition nodes generated
    Paths result = new Paths();
    Rules predecessors = reachabilityIndex.get(c);
    for (Rule r : predecessors) {
        // Optimitization:
        // Given a goal g, if there exists a rule r: c -> g where c is the
        // sole precondition, then when we consider another rule
        // r': c1,c2,...,cn -> g, we are sure that any complex attack
        // discovered through the exploration of the child node r' that
        // requires c in some of its rules is going to be covered by another
        // complex attack discovered through the exploration of the node r
        // This is what the following lines do:
        //
        // C = set of conditions such that
        //     for each c' in C there exists r: c' -> g
        // X = X `union` C \ P
        // where P = {} if r has more than 1 preconditions
        //         = {c"} if c" is the sole precondition of r
        if (isGoal) {
            excludedCondition = Set.union(excludedCondition,
                    singlePreconditions(c, reachabilityIndex));
            if (r.getPreconditions().length == 1)
                excludedCondition.exclude(r.getPreconditions()[0]);
        }
        // end optimization

        // p = r.preconditions\(init `union` includedCondition)
        int[] p = Set.exclude(includedCondition,
                Set.exclude(initConditions, r.getPreconditions()));

        if (p.length == 0)
            result.sum(new Paths(new Set(new int[]{r.getID()})));
        // else if p is subset of reachableConditions && p `intersect` X = 0
        else if (reachableConditions.contains(p) &&
                excludedCondition.emptyIntersection(p)) {
```

```
            Paths subpaths = new Paths();
            for (int i = 0; i < p.length; i++) {
                // I = I `union` p \ {p[i]}
                Set included = Set.union(includedCondition, p);
                included.exclude(p[i]);

                subpaths = Paths.product(subpaths, findAttackPaths(
                        p[i],
                        Set.union(excludedCondition, p[i]), //X `union` {p[i]}
                        included,
                        initConditions,
                        reachableConditions,
                        reachabilityIndex,
                        false));
                if (subpaths == Paths.bottom)
                    break;
            }
            // implementation of line 12 of the pseudocode
            subpaths = Paths.cons(subpaths,r.getID());
            result.sum(subpaths);
        }
    } // end for
    if (result.isEmpty())
        return Paths.bottom;
    else
        return result;
}

/**
 * @param outfile the filename of the intermediate output file
 * @param attackSet the Complex Attack Set
 * @param costs an array of countermeasure costs
 * @param numOfRules the total number of rules
 * @param maxCost the sum of the cost of all countermeasures, for the
 *        definition of the objective function
 * @throws IllegalArgumentException if failed to create the output file
 */
private static void createModelInCPLEXLP(String outfile, Paths attackSet,
        Double[] costs, int numOfRules, Double maxCost) {
    if (outfile == null)
        throw new IllegalArgumentException("error in writing to the " +
                "intermidate output file " + outfile);

    BufferedWriter out;
    try {
        FileWriter fstream = new FileWriter(outfile);
```

```
    out = new BufferedWriter(fstream);

    //Minimize
    // obj: sum -x1 -x2 -x3 -x4
    // Subject To -x1-x2... >= -b
    out.append("Minimize\n");
    out.append(" obj: sum ");
    String objectiveFunctionParts = "";
    Double[] cost = costs;
    for (int i = 1; i <= numOfRules; i++)
        objectiveFunctionParts += " - " + cost[i-1] + " x" + i;
    out.append(objectiveFunctionParts);

    out.append("\nSubject To\n");
} catch (Exception e) {
    throw new IllegalArgumentException("error in writing to the "
            + "intermediate output file " + outfile); }

int constraintID = 1; // constraintID of the current rule

// whether the attack is used in at least one path
boolean[] attacks = new boolean[numOfRules];
for (int i = 0; i < attacks.length; i++) attacks[i] = false;

if (!attackSet.isEmpty()) {
    for (Set s : attackSet) {
        try {
            String output = " c" + constraintID + ": ";
            int numElementaryAttacks = 0; // for calculating b
            int[] rules = s.getElements();
            if (rules != null) {
                for (int i = 0; i < rules.length; i++) {
                    output += "- x" + rules[i];
                    attacks[rules[i]-1] = true;
                    numElementaryAttacks++;
                }
                constraintID++;
                out.append(output + " >= - " +
                    (numElementaryAttacks == 0? 0: numElementaryAttacks-1)
                     + "\n");
            }
        } catch (Exception e) {
            throw new IllegalArgumentException("error in writing to the "
                    + "intermediate output file " + outfile);
        }
    }
```

```
            //System.out.println("no of constraints: " + (constraintID-1));
        }
        try {
            out.append("Bounds\n");
            out.append(" sum = " + maxCost + "\n");
            for (int i = 0; i < attacks.length; i++)
                if (attacks[i] == false)
                    out.append(" x" + (i+1) + " = 1\n");
            out.append("Binary\n");
            for (int i = 0; i < attacks.length; i++)
                if (attacks[i])
                    out.append(" x" + (i+1) + "\n");
            out.append("End\n");
            out.close();
        } catch (Exception e) {
        }finally {
            if (out != null)
                try { out.close();}
                catch (IOException e) {e.printStackTrace();}
        }
    }

    /*
     * @param attackability Given a name of a threat profile, return true if
     *                      there exists a feasible complex attack
     * @param profiles the set of threat profile names
     * @param numOfRules the total number of rules
     * @param countermeasureCosts an array of countermeasure costs, the
     *      countermeasure cost of the rule with ID i is countermeasureCosts[i-1]
     * @param p input parser
     */
    private static String findCountermeasures(
            Hashtable<String,Integer> attackability,
            java.util.Set<String> profiles, int numOfRules,
            Double[] countermeasureCosts, Parser p) {
        String result = "//*************** RESULT *****************//\n";
        double sumOfCountermeasureCost = 0;
        for (String name : profiles) {
            if (attackability.get(name) == ATTACK_POSSIBLE) {
                boolean[] countermeasures = null;
                try {
                     countermeasures = Solve.solve("model(" + name + ").lp",
                             numOfRules);
                } catch (TimeoutException e) { // not used at the moment
                    e.printStackTrace();
                } catch (NoSolutionException e) { // no feasible solution
```

```
            e.printStackTrace();
        } catch (GLPKerrorException e) { // other GLPK errors
            e.printStackTrace();
        }
        result += "# countermeasure(" + name + ") = {";

        // to avoid memory allocation error as Java has problem with the
        // "substring" method when it is applied to long strings
        TreeSet<Integer> v = new TreeSet<Integer>();
        for (int i = 0; i <  countermeasures.length; i++)
            if (countermeasures[i]) {
                // the enumeration of countermeasures begins with 1, not 0
                v.add(new Integer(i+1));
                sumOfCountermeasureCost += countermeasureCosts[i];
            }
        String contromisureString = v.toString();
        result += contromisureString.substring
                    (1,contromisureString.length()-1)+"}";
        //System.out.println("sum of countermeasure cost for " + name +
                //" is " + sumOfCountermeasureCost);

        // compute the subset of counteremeasures such that the sum of
        // their costs is within the budget
        if (p.getBudget(name) != null &&
                sumOfCountermeasureCost > p.getBudget(name)) {
            Log.log("budget exceeded for " + name);
            result += "\n\n// Budget exceeded. Total cost of " +
                    "countermeasures for " + name + " = " +
                    sumOfCountermeasureCost + "\n";
            result += "# countermeasuresWithinBudget(" + name + ") = {";
            result += findCountermeasuresWithinBudgetConstraints
                        (countermeasureCosts,
                         p.getBudget(name),
                         countermeasures,
                         v.size(),
                         sumOfCountermeasureCost)
                     + "}\n";
        }
    }
    else if (attackability.get(name) == NO_ATTACK_POSSIBLE)
        result += "# countermeasure(" + name +
                  ") = {} // no complex attack possible\n";
    else if (attackability.get(name) == NO_POSSIBLE_COUNTERMEASURE)
        result += "# countermeasure(" + name +
         ") = {} // no countermeasure can stop all the complex attacks\n";
}
```

```
    return result;
}

/* compute the subset of countermeasures such that the sum of their costs is
 * within the budget.
 * @param countermeasureCosts an array of countermeasure costs
 * @param budget the budget
 * @param countermeasures characteristic function of the minimal set
 * @param numOfCountermeasures cardinality of the minimal set
 * @param sumOfCountermeasureCost sum of the costs of countermeasures in the
 *          minimal set
 */
private static String findCountermeasuresWithinBudgetConstraints(
        Double[] countermeasureCosts, Double budget,
        boolean[] countermeasures, int numOfCountermeasures,
        double sumOfCountermeasureCost) {
    // contains rule IDs relative to countermeasures in the minimal set
    int[] countermeasuresID = new int[numOfCountermeasures+1];
    // sum of the costs of countermeasures in the subset
    Double sumOfSubsetCountermeasureCost = 0.0; // for statistics only
    // countermeasure costs in the minimal set
    Double[] weight = new Double[numOfCountermeasures+1];
    int j = 1; // index for the array countermeasuresID

    weight[0] = sumOfCountermeasureCost; // use the first position of the
    // array as a temporary placeholder for the sum of countermeasure cost.
    // As that value needs to be normalized as other costs and that the first
    // position of the array is not used.

    // fill countermeasuresID
    for (int i = 0; i < countermeasures.length; i++)
        if (countermeasures[i]) {
            countermeasuresID[j] = i+1; // countermeasure ID, i.e. rule id
            weight[j] = countermeasureCosts[i];
            j++;
        }

    boolean[] subsetOfCountermeasures =
        Knapsack.knapsack(weight, budget, numOfCountermeasures);

    // output formatting
    TreeSet<Integer> v = new TreeSet<Integer>();
    for (int i = 1; i <  subsetOfCountermeasures.length; i++)
        if (subsetOfCountermeasures[i])  {
            v.add(new Integer(countermeasuresID[i]));
            sumOfSubsetCountermeasureCost +=
```

```
                countermeasureCosts[countermeasuresID[i]-1]; }
    //System.out.println("Total cost of countermeasures (within budget): "
    //      + sumOfSubsetCountermeasureCost);
    String result = v.toString();
    return result.substring(1, result.length()-1);
}

/* the reachability index is a data structure that given a condition c,
 * returns the list of rules that have c as their postconditions
 */
private static Index fillReachabilityIndex(Rules rules) {
    Index result = new Index();
    for (Rule r : rules)
        result.add(r.getPostcondition(), r);
    return result;
}

/*
 * Finalize the output file by appending the choice of countermeasures for
 * each threat profile
 * @param outputFileName name of the output file
 * @param result the string to be appended to the output file
 */
private static void finalizeOutputFile(String outputFileName, String result){
    FileWriter foutstream = null;
    try {
        foutstream = new FileWriter(outputFileName, true);
    } catch (IOException e) {
        throw new IllegalArgumentException("the file " + outputFileName +
                " cannot be opened for writing");}
    BufferedWriter out = new BufferedWriter(foutstream);
    try {
        out.append(result);
        out.close();
    } catch (Exception e) {
        Log.error("Error in appending the result of the choice of " +
                "countermeasures to the file " + outputFileName);
    } finally {
        if (out != null)
            try {out.close();}
            catch (IOException e) {e.printStackTrace();}
    }
}

/* Given a condition c, compute the set of conditions S such that
 * forall s in S, there exists a rule r such that s is the sole precondition
```

```
     * and c is the postcondition of r
     */
    private static Set singlePreconditions(int c, Index reachabilityIndex) {
        Rules predecessors = reachabilityIndex.get(c);
        Set result = new Set();
        for (Rule r : predecessors) {
            int[] pre = r.getPreconditions();
            if (pre != null && pre.length == 1)
                result.addElement(pre[0]);
        }
        return result;
    }
}
```

```
package main;

import java.util.Iterator;
import java.util.Vector;

import utils.Set;

/**
 * Paths is a set of sets of integers, which are identifiers of rules
 */
public class Paths implements Iterable<Set>  {
    public static Paths bottom = new Paths();
    private Vector<Set> v = new Vector<Set>();

    Paths() {}

    public Paths(Set set) {
        v.add(set);}

    boolean isEmpty() {
        return v.isEmpty();}

    // Given a set of sets p and a set of sets q, compute a minimal subset set p'
    // such that
    // 1. for each element e1 in p and for each element e2 in q,
    //    there exists an element e3 in p' such that e3 is a subset of the
    //    union of e1 and e2
    // 2. for each element e in p', there exists an element e1 in p and an element
    //    e2 in q such that e is a subset of the union of e1 and e2
    // In other words, it implements the subset product operator
    // The inputs are unmodified
```

```
static Paths product(Paths p, Paths q) {
    if (q == bottom || p == bottom)
        return bottom;
    if (p.isEmpty())
        return q;
    else if (q.isEmpty())
        return p;
    else {
        Paths result = new Paths();
        for (Set original : p.v)
            for (Set s : q)
                result.sum(Set.union(original, s));
        return result; }
}

// 1. s is added to "this" if it is not a superset of any existing element
// 2. all the elements in "this" that are proper superset of s are removed.
// The above conditions guarantees that "this" remains a minimal subset set
private void sum(Set s) {
    Vector<Set> toBeRemoved = new Vector<Set>();
    for (Set s2 : v) {
        if (s2.contains(s.getElements()))
            toBeRemoved.add(s2);
        else if (s.contains(s2.getElements()))
            return;
    }
    v.add(s);
    for (Set del : toBeRemoved)
        v.remove(del);
}

// Add a minimal subset set p to "this" such that for all s in p
// 1. s is added to "this" if it is not a superset of any existing element
// 2. all the elements in "this" that are proper superset of s are removed.
// REQUIRES: this != bottom
public void sum(Paths p) {
    assert(this != bottom);
    if (p != bottom && !p.isEmpty())
        for (Set s : p.v)
            sum(s);
}

// Given a set of sets p and an integer r, compute p' such that
// p' = {e `union` {r} : e in p}
public static Paths cons(Paths p, int r) {
    Paths result = new Paths();
```

```
        if (p == bottom) // special case
            return p;
        else if (p.isEmpty()) // special case
            return new Paths(new Set(new int[]{r}));
        else
            for (Set s : p.v)
                result.sum(Set.union(s, r));
                // we could have used result.v.add(Set.union(s, r)) above
        return result;
    }

    public Iterator<Set> iterator() {
        return v.iterator(); }

    public String toString() {
        if (this == bottom) return "bottom";
        String result = "{";
        for (Set s : v)
            result += s + ",";
        return (v.isEmpty()? result : result.substring(0, result.length()-1))+"}";
    }
}
```

```
package main;

import input.Rule;

import java.util.Hashtable;

/***
 * An index that maps rules to conditions.
 * Given a condition c, it computes the set of rules that have c as postcondition
 *
 * It can be represented logically as:
 * c1: r1, r3, r4
 * c2: r1, r2
 * c3: r8
 *  ...
 */
public class Index {
    private Hashtable<Integer,Rules> t;

    Index() {
        t = new Hashtable<Integer,Rules>(); }
```

```
    /**
     * Add a new rule to the list associated with the condition. If the condition
     * is not already present in the index, a new list will be created with r as
     * the sole element in the list
     * REQUIRES: 0 < condition && r != null
     * @param condition a condition
     * @param r a rule
     * @throws IllegalArgumentException if the precondition is violated
     */
    void add(int condition, Rule r) {
        if (condition <= 0 || r == null)
            throw new IllegalArgumentException("invalid condition or rule: " +
                    "condition = " + condition + "; rule = " + r);
        if  (t.containsKey(new Integer(condition)))
            t.get(new Integer(condition)).insert(r);
        else {
            Rules tmp = new Rules();
            tmp.insert(r);
            t.put(new Integer(condition), tmp);
        }
    }

    /**
     * Get the list of rules associated with the condition c
     * REQUIRES: condition > 0
     * @param c the condition
     * @return the list of rules that have the condition c as postcondition
     *         null if no rule has c as postcondition
     * @throws IllegalArgumentException if the precondition is violated
     */
    Rules get(int c) {
        if (c <= 0) throw new IllegalArgumentException("invalid condition: " + c);
        Rules result = t.get(new Integer(c));
        return result;
    }

    public String toString() {
        return t.toString(); }
}
```

```
package main;

import input.Rule;

import java.util.Iterator;
```

```
import java.util.LinkedList;

/**
 * A list of rules
 */
public class Rules implements Iterable<Rule> {
    private LinkedList<Rule> list = new LinkedList<Rule>();

    /**
     * Add a rule to the current collection of rules
     * REQUIRES: r != null
     * @param r the rule to be added
     * @throws IllegalArgumentException if r == null
     */
    public void insert(Rule r) {
        if (r == null)
            throw new IllegalArgumentException
                ("the object null cannot be added to the list of rules");
        list.addFirst(r); }

    /**
     * Return the total number of rules
     * @return the total number of rules
     */
    public int size() {
        return list.size(); }

    public String toString() {
        String output = "";
        for (Rule r : this)
            output += r + "\n";
        return output; }

    /***
     * The iterator iterates through the elements in a LIFO order.
     */
    public Iterator<Rule> iterator() {
        return new RulesIterator(this); }

    public class RulesIterator implements Iterator<Rule>{
        int nextIndex = 0;
        Rules rules;

        private RulesIterator(Rules rules) {
            assert rules != null;
            this.rules = rules; }
```

```
        public boolean hasNext() {
            return nextIndex < rules.list.size(); }

        public Rule next() {
            Rule result = rules.list.get(nextIndex);
            nextIndex++;
            return result; }

        public void remove() { /* no effect */ }
    }
}
```

B.3. The Countermeasure Selector

```
package riskMitigation;

import org.gnu.glpk.GlpkSolver;

/**
 * Implementation of the Countermeasure Selector component
 */
public class Solve {
    /**
     * Solve a MIP problem saved in the CPLEX LP format
     * i.e. compute the minimal set of countermeasures
     * In the current implementation the problem is solved to optimality using the
     * built-in branch-and-cut method using continous relaxation. No callback
     * method is defined, see Chapter 5 of the documentation of GLPK for details.
     * @param model the name of the input file, which should be located at the
     *   base dir (if it is not the case, specify the full path of the input file)
     * @param numCol the total number of variables. In this case, the total number
     *       of countermeasures.
     * @return An array of boolean of length numCol. Element i is true if the
     *       corresponding countermeasure is in the minimal set.
     *       Return null if the MIP problem is not solved to optimality.
     * @throws TimeoutException if the time limit for the solver is exceeded
     *      (currently no time limit is set)
     * @throws NoSolutionException if there is no feasible solution, in the
     *      current implementation there is always a solution (the one with all
     *      the countermeasures selected)
     * @throws GLPKerrorException other GLPK errors
     */
    public static boolean[] solve(String model, int numCol)
        throws TimeoutException, NoSolutionException, GLPKerrorException {
        // no check is implemented to ensure that the intermediate output
```

```
    // file has not been modified externally
    GlpkSolver solver = GlpkSolver.readCpxlp(model);

    // optional tweaks for the solver, can be commented out entirely
    // see documentation of GLPK constant for more options
    solver.setIntParm(GlpkSolver.LPX_K_PRESOL, 1);
    solver.setIntParm(GlpkSolver.LPX_K_DUAL, 1);

    /* set timeout (in seconds)
       -1 is the default value, which means no time limit */
    //solver.setRealParm(GlpkSolver.LPX_K_TMLIM, -1);

    /* create a copy of the formulated problem in the MPS format, it may be
     * useful if you want to use other solvers instead of GLPK */
    // solver.writeMps(model+".mps");

    int simplexReturnCode = solver.simplex(); // LP relaxation
    switch (simplexReturnCode) {
        case GlpkSolver.LPX_E_TMLIM:
            throw new TimeoutException("GLPK time limit exhausted");
        case GlpkSolver.LPX_E_NOFEAS:
            throw new NoSolutionException("GLPK: No feasible(real) solution");
        case GlpkSolver.LPX_E_INSTAB:
            throw new GLPKerrorException("GLPK: numerical instability");
    }
    int integerReturnCode = solver.integer();
    switch (integerReturnCode) {
        case GlpkSolver.LPX_E_TMLIM:
           throw new TimeoutException("GLPK time limit exhausted");
        case GlpkSolver.LPX_E_NOFEAS:
           throw new NoSolutionException("GLPK:No feasible integer solution");
        case GlpkSolver.LPX_E_INSTAB:
           throw new GLPKerrorException("GLPK: numerical instability");
        case GlpkSolver.LPX_E_FAULT:
           throw new GLPKerrorException("GLPK: not possible to start the " +
                "search of integer solution, possibly timeout");
    }
    if (simplexReturnCode == GlpkSolver.LPX_E_OK &&
            integerReturnCode == GlpkSolver.LPX_E_OK) {
            boolean[] result = new boolean[numCol];
            for (int i = 0; i < result.length; i++)
                result[i] = solver.mipColVal(i+2) <= 0;
            return result;
        }
    return null; // if there is any error
}
```

```
}
```

```
package riskMitigation;

/**
 * Selection of the subset of the minimal set such that the total costs
 * of countermeasures in the subset does not exceed the budget
 */
public class Knapsack {
    /** Based on an algorithm in the book
     * "Introduction to Programming in Java: An Interdisciplinary Approach"
     * @param weight costs of the countermeasures in the minimal set
     * @param capacity the budget
     * @param numItems cardinality of the minimal set
     */
    public static boolean[] knapsack(Double[] weight, Double capacity,
            int numItems) {
        int[] normalizedWeight = normalize(weight); // convert costs to integers
        int sumOfWeight = normalizedWeight[0];
        normalizedWeight[0] = 0; // reset the value as we have used that place as
                                 // a temporary place holder
        int normalizedCapacity =
            new Double((sumOfWeight/weight[0])*capacity).intValue();

        // in this case, we have defined the profit the same as the cost
        // however, other choices are possible as discussed in the thesis
        int[] profit = normalizedWeight;

        int[][] opt = new int[numItems+1][normalizedCapacity+1];
        boolean[][] sol = new boolean[numItems+1][normalizedCapacity+1];

        for (int n = 1; n <= numItems; n++)
            for (int w = 1; w <= normalizedCapacity; w++) {
                // don't take item n
                int option1 = opt[n-1][w];

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (normalizedWeight[n] <= w) option2 =
                    profit[n] + opt[n-1][w-normalizedWeight[n]];

                // select better of two options
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
```

```
        // determine which items to take
        boolean[] take = new boolean[numItems+1];
        for (int n = numItems, w = normalizedCapacity; n > 0; n--) {
            if (sol[n][w]) { take[n] = true;  w = w - normalizedWeight[n]; }
            else           { take[n] = false;                              }
        }
        return take;
    }

    /*
     * Given an array of doubles, convert it to an array of integers by
     * multiplying each element by 10^m where
     * m is the maximum number of decimal places
     */
    private static int[] normalize(Double[] d) {
        String[] str = new String[d.length];
        int maxNumOfDecimalPlaces = 0;
        for (int i = 0; i < d.length; i++) {
            str[i] = d[i].toString();
            int whereIsDecimalPt = str[i].indexOf('.');
            if (whereIsDecimalPt != -1 &&
                    Integer.parseInt(str[i].substring(whereIsDecimalPt+1)) != 0) {
                int numDecimalDigits = str[i].length()-whereIsDecimalPt-1;
                if (numDecimalDigits > maxNumOfDecimalPlaces)
                    maxNumOfDecimalPlaces = numDecimalDigits;
            }
        }
        int[] result = new int[d.length];
        for (int i = 0; i < d.length; i++)
            result[i] = (new Double(d[i] *
                    Math.pow(10,maxNumOfDecimalPlaces))).intValue();
        return result;
    }
}
```

Bibliography

[1] Common vulnerabilities and exposures dictionary. http://cve.mitre.org

[2] CPLEX LP Format http://www.ilog.com/products/cplex/product/represent.cfm

[3] GLPK. http://www.gnu.org/software/glpk/

[4] Nessus Scanner. http://www.nessus.org

[5] NuSMV. NuSMV: A New Symbolic Model Checker. http://afrodite.itc.it:1024/~nusmv/.

[6] SANS Top-20 2007 Security Risks (2007 Annual Update) http://www.sans.org/top20/2007/top20.pdf

[7] P. Ammann, J.Pamula, R.Ritchey and J.Street. A host-based approach to network attack chaining analysis. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 72-84. IEEE Computer Society, 2005.

[8] P.Ammann, D.Wijesekera and S.Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of 9th ACM Conference on Computer and Communications Security*, pages 217-224. ACM Press, 2002.

[9] F. Baiardi, M. Pioli, S. Suin and C. Telmon, Assessing the Risk of an Information Infrastructure through Security Dependencies, *First Workshop on Critical Information Infrastructure Security*, Samo, August 2006.

[10] F. Baiardi, C. Telmon and D. Sgandurra, Hierarchical, Model-Based Risk Management of Critical Infrastructures. Reliability Engineering and System Safety, Elsevier Press, to appear.

[11] M. Freiss. Protecting Networks with SATAN, O'Reilly, 1999.

[12] K. Ingols, R. Lippmann and K. Piwowarski. Practical Attack Graph Generation for Network Defense. In *Proceedings of the 22nd Annual Computer Security Applications Conference.*

[13] S. Jajodia, S. Noel and B. O'Berry. Topological analysis of network attack vulnerability. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, Managing Cyber Threats: Issues, Approaches and Challanges, chapter 5. Kluwer Academic Publisher, 2003.

[14] S. Jha, O. Sheyner and J. Wing, Two Formal Analysis of Attack Graphs, *15th IEEE Computer Security Foundations Workshop*, p.49, June 2002.

[15] P. Meil, T. Grance et al. NVD national vulnerability database. http://nvd.nist.gov.

[16] R. P. Lippmann and K. W. Ingols. An annotated review of past papers on attack graphs. Technical report, MIT Lincoln Laboratory, Lexington, MA, 2005. ESC-TR-2005-054.

[17] R.P. Lippmann et al. Evaluating and strengthening enterprise network security using attack graphs. Technical report. MIT Lincoln Laboratory. Lexington, MA, 2005. ESC-TR-2005-064.

[18] X. Ou, W. F. Boyer and M. A. McQueen. A Scalable Approach to Attack Graph Generation. 2006

[19] C. Philips and L. P. Swiler. A graph-based system for network vulnerability analysis. In *Proceedings of the 1998 workshop on New security paradigms*, p.71-79, September 22-26, 1998, Charlottesville, Virginia, United States.

[20] R. W. Ritchey and P. Ammanna. Using model checking to analyze network vulnerabilites. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 156-165, Oakland, CA, May 2000.

[21] R. Sedgewick, K. Wayne. Introduction to Programming in Java: An Interdisciplinary Approach. Addison Wesley, 2007.

[22] O.Sheyner, J.Haines, S.Jha, R.Lippmann and J. Wing. Automated generation and analysis of attack graphs. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2002

[23] O. Sheyner and J. Wing. Tools for generating and analyzing attack graphs. FMCO 2003, LNCS 3188, pp. 344–371, 2004.

[24] Swarup, Jajodia, Pamula. Rule-based Topological Vulnerability Analysis. In Computer Network Security, selected papers from the 3rd International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, 2005.

[25] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.