The ISC Int'l Journal of Information Security

ISeCure

January 2010, Volume 2, Number 1 (pp. 13–32)

http://www.isecure-journal.org

A Particle Swarm Optimization Algorithm for Minimization Analysis of Cost-Sensitive Attack Graphs

Mahdi Abadi [a,*], Saeed Jalili [a]

[a] *Faculty of Electrical and Computer Engineering, Tarbiat Modares University, Tehran, Iran.*

ARTICLE INFO.

Article history:
Received: 19 April 2009
Revised: 6 November 2009
Accepted: 13 December 2009
Published Online: 26 January 2010

Keywords:
Particle Swarm Optimization, Attack Scenario, Countermeasure, Cost-Sensitive Attack Graph, Minimization Analysis

ABSTRACT

To prevent an exploit, the security analyst must implement a suitable countermeasure. In this paper, we consider cost-sensitive attack graphs (CAGs) for network vulnerability analysis. In these attack graphs, a weight is assigned to each countermeasure to represent the cost of its implementation. There may be multiple countermeasures with different weights for preventing a single exploit. Also, a single countermeasure may prevent multiple exploits. We present a binary particle swarm optimization algorithm with a time-varying velocity clamping, called SwarmCAG-TVVC, for minimization analysis of cost-sensitive attack graphs. The aim is to find a critical set of countermeasures with minimum weight whose implementation causes the initial nodes and the goal nodes of the graph to be completely disconnected. This problem is in fact a constrained optimization problem. A repair method is used to convert the constrained optimization problem into an unconstrained one. A local search heuristic is used to improve the overall performance of the algorithm. We compare the performance of SwarmCAG-TVVC with a greedy algorithm GreedyCAG and a genetic algorithm GenNAG for minimization analysis of several large-scale cost-sensitive attack graphs. On average, the weight of a critical set of countermeasures found by SwarmCAG-TVVC is 6.15 percent less than the weight of a critical set of countermeasures found by GreedyCAG. Also, SwarmCAG-TVVC performs better than GenNAG in terms of convergence speed and accuracy. The results of the experiments show that SwarmCAG-TVVC can be successfully used for minimization analysis of large-scale cost-sensitive attack graphs.

1 Introduction

Our society has become increasingly dependent on computer networks and the trend towards larger networks will continue. Each network host runs different software packages and supports several modes of connectivity. Despite the best efforts of software architects and developers, network hosts inevitably contain a number of vulnerabilities. Hence, it is not feasible for a network administrator to remove all vulnerabilities present in the network hosts. Therefore, the recent focus in security of such networks is on analysis of vulnerabilities globally, finding exploits that are more critical, and preventing them to thwart an intruder.

* Corresponding author.
Email addresses: abadi@modares.ac.ir (M. Abadi), sjalili@modares.ac.ir (S. Jalili)
ISSN: 2008-2045

When evaluating the security of a network, it is rarely enough to consider the presence or absence of isolated vulnerabilities. This is because intruders often combine exploits against multiple vulnerabilities in order to reach their goals [1]. For example, an intruder might exploit the vulnerability of a particular version of FTP to overwrite the ".rhosts" file on a victim host. In the next step, the intruder could remotely log in to the victim. In a subsequent step, the intruder could use the victim host as a base to launch another exploit on a new victim.

In this paper, we consider cost-sensitive attack graphs (CAGs) for network vulnerability analysis. In these attack graphs, a weight is assigned to each countermeasure to represent the cost of its implementation. There may be multiple countermeasures with different weights for preventing a single exploit. Also, a single countermeasure may prevent multiple exploits. The aim of minimization analysis of cost-sensitive attack graphs is to find a critical set of countermeasures with minimum weight so that by implementing them an intruder cannot reach his goals using any attack scenario. We present a binary particle swarm optimization algorithm with a time-varying velocity clamping, called SwarmCAG-TVVC, for minimization analysis of cost-sensitive attack graphs. We compare the performance of SwarmCAG-TVVC with a greedy algorithm GreedyCAG and a genetic algorithm GenNAG [2] for minimization analysis of several large-scale cost-sensitive attack graphs. We also show the effect of different settings of parameters on the performance of SwarmCAG-TVVC.

Particle swarm optimization (PSO) is a population based stochastic optimization algorithm developed by Kennedy and Eberhart [3]. PSO has proved to be efficient at solving engineering problems. SwarmCAG-TVVC extends the binary PSO for minimization analysis of cost-sensitive attack graphs by incorporating a time-varying velocity clamping, a greedy repair algorithm, a redundant countermeasure elimination algorithm, and a local search heuristic.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 provides an overview of PSO. Section 4 introduces our network security model and Section 5 describes the process of minimization analysis of cost-sensitive attack graphs. Section 6 presents GreedyCAG and Section 7 presents SwarmCAG-TVVC. Section 8 reports the experimental results, and finally Section 9 draws some conclusions.

2 Related Work

Phillips and Swiler [4] proposed the concept of attack graphs, where each node represents a possible attack state. Edges represent a change of state caused by a single action taken by the intruder. In fact, attack graphs represent the ways in which an intruder can exploit vulnerabilities to break into a system [5].

Sheyner *et al.* [6] used a modified version of the model checker NuSMV [7] to produce attack graphs. Ammann *et al.* [8] introduced a monotonicity assumption and applied it to develop a polynomial algorithm to encode all edges of an attack graph without actually computing the graph itself. These attack graphs are essentially similar to [5], where any path in the graph from an initial node to a goal node shows a sequence of exploits that an intruder can launch to reach his goal.

Noel *et al.* [9] presented a number of techniques for managing attack graph complexity through visualization. Mehta *et al.* [10] presented a ranking scheme for the nodes of an attack graph. Rank of a node shows its importance based on factors such as the probability of an intruder reaching that node. Given a ranked attack graph, the system administrator can concentrate on relevant subgraphs to figure out how to start deploying security measures.

Noel *et al.* [11] and Wang *et al.* [12] proposed a solution to automate the task of hardening a network against attack scenarios. They first represented given critical resources as a logic proposition of initial conditions. They then simplified the proposition to make hardening options explicit. Among the options, they finally chose a solution with minimum cost based on given assumptions on the cost of initial conditions. They presented a procedure for choosing a minimum cost solution, but it has an unavoidable exponential worst-case complexity [12]. In a large enterprise network, there can be many initial conditions because these include vulnerabilities on each host, reachability between all host pairs, and trust relationships between hosts.

Ou *et al.* [13] presented logical attack graphs, which directly illustrate logical dependencies among attack goals and configuration information. Their attack graph generation tool builds upon MulVAL [14], a network security analyzer based on logic programming.

The aim of minimization analysis of simple attack graphs is to find a minimum critical set of exploits that completely disconnect the initial nodes and the goal nodes of the graph. Sheyner *et al.* [6] and Jha *et al.* [15, 16] showed this problem is in fact *NP*-hard. They presented an approximation algorithm, which is able to find an approximately-optimal set of exploits,

which must be prevented to thwart an intruder. Abadi and Jalili [2, 17] presented a genetic algorithm and a binary particle swarm optimization algorithm for minimization analysis of simple attack graphs. In the above algorithms, the cost of preventing exploits is considered to be the same, while in the real world, the cost of preventing an exploit could be different from the cost of preventing another exploit. For example, in order to prevent an exploit, the security analyst may patch the vulnerability that made this exploit possible, whereas to prevent some other exploits, it may be required to change the firewall configuration or deploy an intrusion detection and prevention system.

While it is currently possible to generate very large and complex attack graphs, relatively little work has been done to analyze them.

3 Particle Swarm Optimization

Particle swarm optimization (PSO) is a population based stochastic optimization. It was inspired by social behavior of flocks of birds when they are searching for food. In PSO, the potential solutions, called *particles*, fly through the problem space exploring for better regions. The position of a particle is influenced by the best position visited by itself and the position of the best particle in its neighborhood. When the neighborhood of a particle is the entire swarm, the best position in the neighborhood is referred to as the global best particle, and the resulting algorithm is referred to as a *gbest* PSO [18].

The performance of each particle is measured using a predefined fitness function, which is related to the problem to be solved. Each particle in the swarm has a current position, x_i, a velocity (rate of position change), v_i, and a personal best position, y_i. The personal best position of particle i shows the best fitness reached by that particle at a given time. Let f be the objective function to be maximized. The personal best position of a particle at iteration or time step t is updated as

$$y_i(t) = \begin{cases} y_i(t-1) & \text{, if } f(x_i(t)) \leq f(y_i(t-1)) \\ x_i(t) & \text{, if } f(x_i(t)) > f(y_i(t-1)) \end{cases} \quad (1)$$

For the *gbest* model, the best particle is determined from the entire swarm by selecting the best personal best position. This position is denoted by $\hat{y}$.

The equation that manipulates the velocity is called the *velocity update equation* and is stated as

$$\begin{aligned} v_{ij}(t+1) = v_{ij}(t) &+ c_1 r_{1j}(t)(y_{ij}(t) - x_{ij}(t)) \\ &+ c_2 r_{2j}(t)(\hat{y}_j(t) - x_{ij}(t)) \end{aligned} \quad (2)$$

where $v_{ij}(t+1)$ is the velocity updated for the jth dimension, $j = 1, 2, ..., d$. c_1 and c_2 are the acceleration constants, where the first moderates the maximum step size towards the best personal of the particle, while the second moderates the maximum step size towards the global best particle in just one iteration. $r_{1j}(t)$ and $r_{2j}(t)$ are two random values in the range $[0, 1]$ and give the PSO algorithm a stochastic search property.

Velocity updates on each dimension can be clamped with a user defined maximum velocity V_{max}, which would prevent them from exploding, thereby causing premature convergence [19, 20].

Each particle updates its position using the following equation:

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (3)$$

Binary Particle Swarm Optimization

Kennedy and Eberhart [21] have adapted PSO to search in binary spaces. For the binary PSO, the elements of x_i, y_i and $\hat{y}$ can only take the values 0 and 1. The velocity v_i is interpreted as a probability to change a bit from 0 to 1, or from 1 to 0 when updating the position of particles. Therefore, the velocity vector remains continuous-valued. Since each v_{ij} is a real value, a mapping needs to be defined from v_{ij} to a probability in the range $[0, 1]$. This is done by using a sigmoid function to squash velocities into a $[0, 1]$ range. The sigmoid function is defined as

$$sig(v) = \frac{1}{1 + e^{-v}} \quad (4)$$

The equation for updating positions is then replaced by the following probabilistic update equation:

$$x_{ij}(t+1) = \begin{cases} 0 & \text{, if } r_{3j}(t) \geq sig(v_{ij}(t+1)) \\ 1 & \text{, if } r_{3j}(t) < sig(v_{ij}(t+1)) \end{cases} \quad (5)$$

where $r_{3j}(t)$ is a random value in the range $[0, 1]$.

In the binary PSO, the velocity is interpreted as a probability of change. Hence, the velocity clamping, v_{max}, sets the minimal probability for a bit to change its value [22].

In this paper, we use the *gbest* model of the binary PSO for minimization analysis of cost-sensitive attack graphs.

4 Network Security Model

We use a network security model similar to that in [6, 15, 16]. Our network security model is a tuple (S, H, C, T, E, M, R), where S is a set of services, H is a set of hosts connected to the network, C is a relation

expressing connectivity between hosts, T is a relation expressing trust between hosts, E is a set of individual exploits that an intruder can use to construct attack scenarios, M is a set of countermeasures that must be implemented to prevent exploits, and R is a model of the intruder.

Services

Each service $s \in S$ is a pair (svn, p), where svn is the service name and p is the port on which the service is listening.

Hosts

Each host $h \in H$ is a tuple $(id, svcs, plvl, vuls)$, where id is a unique host identifier, $svcs$ is a set of services running on the host, $plvl$ is the level of privilege that the intruder has on the host, and $vuls$ is a set of host-specific vulnerable components. For the sake of simplicity, we only consider three privilege levels: *none*, *user*, and *root*.

Network Connectivity

Network connectivity is modeled as a relation $C \subseteq H \times H \times P$, where P is a set of port numbers. Each network connection $c \in C$ is a triple (h_s, h_t, p_t), where h_s is the source host, h_t is the target host, and p_t is the target port number. Note that the connectivity relation incorporates network elements such as firewalls that restrict the ability of one host to connect to another one.

Trust Relationships

Trust relationships are modeled as a relation $T \subseteq H \times H$, where $T(h_t, h_s)$ indicates that a user may log in from host h_s to host h_t without authentication.

Exploits

Each exploit $e \in E$ is a tuple $(pre, h_s, h_t, post)$, where pre is a list of conditions that must hold before launching the exploit, h_s is the host from which the exploit is launched, h_t is the host targeted by the exploit, and $post$ specifies the effects of the exploit on the network. An exploit cannot be launched until all of its required conditions have been satisfied.

Countermeasures

To prevent an exploit $e \in E$, the security analyst must implement a suitable countermeasure $m \in M$, such as

- changing the firewall configuration
- patching the vulnerability that made this exploit possible
- deploying a host-based or a network-based intrusion detection and prevention system
- modifying the configuration of network services and applications
- deleting user accounts
- changing access rights
- setting up a virtual private network (VPN)

A weight is assigned to each countermeasure to represent the cost of its implementation. There may be multiple countermeasures with different weights for preventing a single exploit. Also, a single countermeasure may prevent multiple exploits.

Intruder

The intruder has some knowledge about the target network, such as known vulnerabilities, user passwords, and information gathered with port scans. The intruder's knowledge is modeled as a relation $R \subseteq ID \times PW \times VUL \times INF$, where ID is a set of host identifiers, PW is a set of user passwords, VUL is a set of known vulnerabilities, and INF is a set of information gathered through port scans and operating system identification techniques.

5 Formal Definition of the Problem

Let $E = \{e_1, e_2, ..., e_n\}$ be the set of exploits, $M = \{m_1, m_2, ..., m_p\}$ be the set of countermeasures, and $prv : M \rightarrow 2^E$ be a function. An exploit $e_i \in prv(m_j)$ if and only if implementing the countermeasure m_j prevents the exploit e_i.

Definition 1. For each countermeasure $m_j \in M$, the degree $deg(m_j)$ is defined to be the number of exploits that are prevented by implementing m_j.

With each exploit $e_i \in E$, a set of countermeasures $ms(e_i)$ is associated.

$$ms(e_i) = \{m_j \in M \mid e_i \in prv(m_j)\} \qquad (6)$$

Definition 2. A cost-sensitive attack graph is a tuple $G = (V, A, V_0, V_f, L)$, where V is the set of nodes, A is the set of directed edges, $V_0 \subseteq V$ is the set of initial nodes, $V_f \subseteq V$ is the set of goal nodes, and $L : A \rightarrow E \times 2^M$ is a labeling function, where $L(a) = (e_i, ms(e_i))$ if and only if an edge $a = (v, v')$ corresponds to an exploit e_i with which a set of countermeasures $ms(e_i)$ is associated.

A path π in G is a sequence of nodes $v_1, v_2, ..., v_{r+1}$, such that $v_i \in V$ and $(v_i, v_{i+1}) \in A$, where $1 \leq i \leq r$. The label of path π is denoted by

$$L(\pi) = (e_1, ms(e_1)), (e_2, ms(e_2)), ..., (e_r, ms(e_r))$$
$$\text{where}$$
$$(e_i, ms(e_i)) = L(v_i, v_{i+1}), \quad 1 \leq i \leq r \qquad (7)$$

Each attack scenario corresponds to a complete path that starts from an initial node and ends in a goal node.

Let $S = \{s_1, s_2, ..., s_l\}$ be the set of attack scenarios represented by the cost-sensitive attack graph G. The attack scenario $s_k \in S$ defines a set of countermeasures $ms(s_k)$.

$$ms(s_k) = \bigcup_{e_i \in s_k} ms(e_i) \tag{8}$$

Definition 3. The attack scenario $s_k \in S$ is hit by the exploit $e_i \in E$ if $e_i \in s_k$.

Definition 4. The attack scenario $s_k \in S$ is prevented by implementing the countermeasure $m_j \in M$ if $m_j \in ms(s_k)$.

Definition 5. For each countermeasure $m_j \in M$, the total prevention value $pv_t(m_j)$ is defined to be the number of attack scenarios that are prevented by implementing m_j.

$$pv_t(m_j) = |\{s_k \in S \mid m_j \in ms(s_k)\}| \tag{9}$$

Definition 6. Let $U \subseteq M$ be a subset of countermeasures and $ps(U)$ be the set of attack scenarios prevented by implementing the countermeasures in U.

$$ps(U) = \{s_k \in S \mid m_j \in ms(s_k) \text{ for some } m_j \in U\} \tag{10}$$

A countermeasure $m_j \in M$ is called redundant with respect to U if $ps(U \backslash \{m_j\}) = ps(U)$.

Definition 7. For each countermeasure $m_j \notin U$, the partial prevention value $pv_r(m_j, U)$ is defined to be the number of attack scenarios that are prevented by implementing m_j, but that are not prevented by implementing any countermeasure in U.

$$pv_r(m_j, U) = |\{s_k \in S \mid m_j \in ms(s_k) \wedge s_k \notin ps(U)\}| \tag{11}$$

Definition 8. Let W be the set of weights and $w : M \rightarrow W$ be a function that assigns a weight to each countermeasure. The weight of the set of countermeasures U, denoted by $w_s(U)$, is the total weight of countermeasures that are members of U.

$$w_s(U) = \sum\nolimits_{m_j \in U} w(m_j) \tag{12}$$

Definition 9. A subset of countermeasures $M_c \subseteq M$ is critical if and only if all attack scenarios are prevented by implementing the countermeasures in it. Equivalently, M_c is critical if and only if every complete path from an initial node to a goal node of the network attack graph has at least one edge labeled with an exploit $e_i \in es(M_c)$, where $es(M_c)$ is the set of exploits prevented by implementing the countermeasures in M_c.

$$es(M_c) = \bigcup_{m_j \in M_c} prv(m_j) \tag{13}$$

Definition 10. The critical set of countermeasures M_c is minimal if it contains no redundant countermeasure.

Definition 11. The critical set of countermeasures M_c has the minimum weight if there is no critical set of countermeasures M'_c such that

$$w_s(M'_c) < w_s(M_c) \tag{14}$$

Definition 12. For each countermeasure $m_j \in M_c$, the exclusive prevention value $pv_x(m_j, M_c)$ is defined to be the number of attack scenarios that are prevented by implementing m_j, but that are not prevented by implementing any countermeasure in $M_c \backslash \{m_j\}$.

$$pv_x(m_j, M_c) = |\{s_k \in S \mid m_j \in ms(s_k) \wedge s_k \notin ps(M_c \backslash \{m_j\})\}| \tag{15}$$

Definition 13. The countermeasure $m_j \in M_c$ is called *candidate redundant* with respect to M_c if $pv_x(m_j, M_c) = 0$.

The aim of minimization analysis of a cost-sensitive attack graph is to find a critical set of countermeasures with minimum weight. In general, there can be multiple critical set of countermeasures with minimum weight.

A typical process for minimization analysis of a cost-sensitive attack graph is shown in Figure 1. First, vulnerability scanning tools, such as Nessus [23], determine vulnerabilities of individual hosts. Using this vulnerability information along with exploit templates, intruder's goals, and other information about the network, such as connectivity between hosts, a cost-sensitive attack graph is generated. In this directed graph, each complete path from an initial node to a goal node corresponds to an attack scenario. The minimization analysis of the cost-sensitive attack graph determines a critical set of countermeasures with minimum weight that must be implemented to guarantee the intruder cannot reach his goals using any attack scenarios.

6 GreedyCAG

In this section, we present GreedyCAG, a greedy algorithm for minimization analysis of cost-sensitive attack graphs. GreedyCAG is in fact an approximation algorithm, which is only able to find an approximately-optimal critical set of countermeasures for a given cost-sensitive attack graph.

Let M be the set of countermeasures and S be the set of attack scenarios represented by the cost-sensitive attack graph G. At each step, GreedyCAG chooses a countermeasure $m_k \in M$ that maximizes the ratio between the total prevention value $pv_t(m_k)$

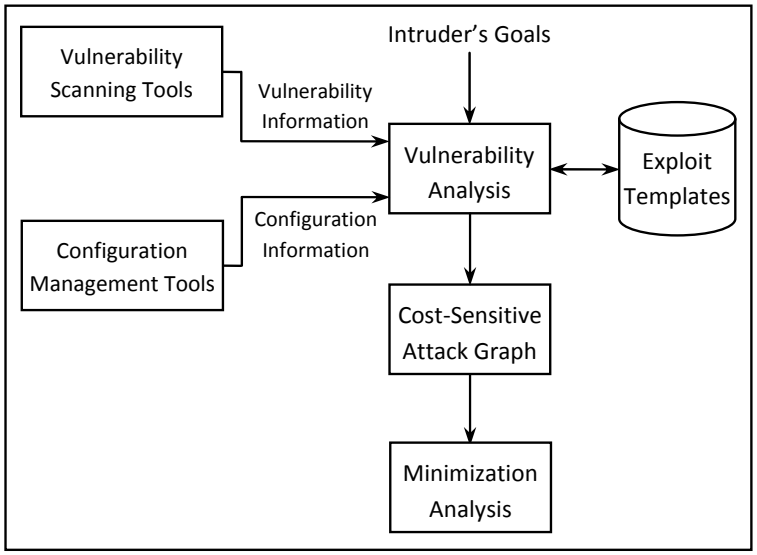

Figure 1. Minimization analysis of cost-sensitive attack graphs.

Procedure GreedyCAG(M, S)

$M_c = \varnothing$
while $S \neq \varnothing$ **do**
  for all countermeasure $m_j \in M$ **do**
    Compute $pv_t(m_j)$
  end for
  Choose a countermeasure $m_k \in M$ that maximizes the ratio $pv_t(m_k)/w(m_k)$
  $M_c = M_c \cup \{m_k\}$
  $S = S \setminus ps(\{m_k\})$
  $M = M \setminus \{m_k\}$
end while
Eliminate redundant countermeasures of M_c
return M_c

Figure 2. The GreedyCAG algorithm.

of the countermeasure and its weight $w(m_k)$. The attack scenarios that are prevented by implementing this countermeasure are not considered for the future steps. This is repeated until a critical set of countermeasures M_c is obtained. After that, redundant countermeasures of M_c are eliminated. Figure 2 shows the pseudo-code of GreedyCAG.

6.1 Redundant Countermeasure Elimination Algorithm

The critical set of countermeasures generated by GreedyCAG may contain redundant countermeasures, which must be eliminated.

Let M_c be the critical set of countermeasures generated by GreedyCAG and R be the set of candidate redundant countermeasures of M_c.

$$R = \{m_j \in M_c \mid pv_x(m_j, M_c) = 0\} \qquad (16)$$

Procedure Elimination(M_c)

$R = \{m_j \in M_c | pv_x(m_j, M_c) = 0\}$
while $R \neq \varnothing$ **do**
  Choose a countermeasure $m_k \in R$ that maximizes the redundant value $rv(m_k, M_c)$
  $M_c = M_c \setminus \{m_k\}$
  $R = \{m_j \in M_c | pv_x(m_j, M_c) = 0\}$
end while
return M_c

Figure 3. The algorithm for eliminating redundant countermeasures from critical sets.

For each candidate redundant countermeasure $m_j \in R$, the *redundant value* $rv(m_j, M_c)$ is defined as

$$rv(m_j, M_c) = \frac{w(m_j)}{(1 + \sum_{m_k \in M_c \setminus \{m_j\}} pv_x(m_k, M_c \setminus \{m_j\}))} \qquad (17)$$

The redundant value $rv(m_j, M_c)$ is used to evaluate candidate redundant countermeasures of M_c in order to choose a candidate redundant countermeasure to be removed from it.

Figure 3 shows an algorithm used to eliminate redundant countermeasures of M_c. The algorithm is based on the idea that it is good to remove a candidate redundant countermeasure from M_c if it has a large weight and prevents attack scenarios that are prevented by too many other countermeasures in M_c. Hence, at each step, the algorithm chooses and removes a candidate redundant countermeasure m_k from M_c that maximizes the redundant value $rv(m_k, M_c)$. This is repeated until a minimal critical set of countermeasures is obtained.

Procedure SwarmCAG-TVVC

Set parameters, create and initialize the swarm, and set the initial velocities to zero
while termination condition not met **do**
  for all particle i **do**
    if x_i does not represent a critical set **then**
      Apply the greedy repair algorithm to x_i
    end if
    Eliminate redundant countermeasures of x_i
    if $U(0,1) < P_l$ **then**
      Apply the local search heuristic to x_i
    end if
    Update the personal best position y_i using equation (1)
  end for
  Update the global best position $\hat{y}$
  Update the velocity clamping V_{max} using equation (21)
  for all particle i **do**
    Update the velocity v_i using equation (2)
    Update the position x_i using equation (5)
  end for
end while

Figure 4. The SwarmCAG-TVVC algorithm.

7 SwarmCAG-TVVC

In this section, we present SwarmCAG-TVVC, a binary particle swarm optimization algorithm with a time-varying velocity clamping for minimization analysis of cost-sensitive attack graphs. The problem of minimization analysis of cost-sensitive attack graphs is in fact a constrained optimization problem in which the objective is to find a solution with minimum weight and the constraint is that the solution must be critical (i.e., it must prevent all attack scenarios).

Figure 4 shows the pseudo-code of the SwarmCAG-TVVC. The first step is to initialize the swarm and control parameters. Then, repeated iterations of the algorithm are executed until some termination condition is met (e.g., a maximum number of iterations is reached). Within each iteration, if each particle's current position x_i does not represent a critical set of countermeasures, a greedy repair algorithm is applied to it. Then, redundant countermeasures of x_i are eliminated. After that, with probability P_l, a local search heuristic is applied to x_i in order to improve it. Then, the particle's personal best position y_i is updated using equation (1). The global best position $\hat{y}$ is then determined from the entire swarm by selecting the best personal best position. Finally, the velocity clamping V_{max} is updated using equation (21) and the velocity and position of each particle are updated using equation (2) and equation (5).

7.1 Problem Representation

Let $M = \{m_1, m_2, ..., m_p\}$ be the set of countermeasures. Each particle position x_i corresponds to an p-bit vector $(x_{i1}, x_{i2}, ..., x_{ip})$ and represents a subset of countermeasures $M_i \subseteq M$ in which the countermeasure $m_j \in M_i$ if and only if the element $x_{ij} = 1$.

$$M_i = \{m_j \in M \mid x_{ij} = 1\} \tag{18}$$

The total weights of countermeasures corresponding to the particle position x_i is

$$w_o(x_i) = w_s(M_i) = \sum\nolimits_{m_j \in M_i} w(m_j) \tag{19}$$

Let $S = \{s_1, s_2, ..., s_l\}$ be the set of attack scenarios represented by the cost-sensitive attack graph G. The attack scenario $s_k \in S$ is prevented by the particle position x_i if $ms(s_k) \cap M_i \neq \varnothing$. Similarly, the attack scenario $s_k \in S$ is prevented by implementing the countermeasure m_j if $m_j \in ms(s_k)$.

The particle position x_i represents a critical set of countermeasures if all attack scenarios are prevented by it. The aim of minimization analysis of cost-sensitive attack graphs is to find a critical set of countermeasures with minimum weight. Hence, SwarmCAG-TVVC uses the following fitness function to evaluate the quality of x_i:

$$\begin{aligned} &f(x_i) = w_t - w_o(x_i), \\ &\text{where } w_t = \sum\nolimits_{m_j \in M} w(m_j) \end{aligned} \tag{20}$$

7.2 Time-Varying Velocity Clamping (TVVC)

In the binary PSO, the meaning and behavior of velocity clamping differ substantially from the real-valued PSO. With the velocity interpreted as a probability of change, velocity clamping, V_{max}, sets the minimal probability for a bit to change its value from 0 to 1, or from 1 to 0 [22]. If V_{max} is a small value, it gives a bigger chance to a bit to change its value (i.e., exploring the search space), while if V_{max} is large, it allows particles to converge on a solution (i.e., exploiting the search space). Accordingly, we use a time-varying velocity clamping,

$$V_{max}(t) = \begin{cases} V_{max}(0) & \text{, if } t \leq t_{tvvc} \\ V_{max}(0) + \Delta V_{max} \cdot \frac{t - t_{tvvc}}{t_{\max} - t_{tvvc}} & \text{, otherwise} \end{cases} \tag{21}$$

where $\Delta V_{\max} = (V_{\max}(t_{\max}) - V_{\max}(0))$, t_{max} is the maximum number of iterations, $V_{max}(0)$ is the initial velocity clamping, $V_{max}(t_{max})$ is the final velocity clamping, and $V_{max}(t)$ is the velocity clamping at iteration t. t_{tvvc} is the starting time of the time-varying velocity clamping.

In Section 8, we will show the effect of the time-

Procedure GreedyRepair(x_i)

1: $M_i = \{m_j \in M | x_{ij} = 1\}$
2: **while** x_i does not represent a critical set **do**
3: Choose a countermeasure $m_k \in M$ such that $m_k \notin M_i$ and it maximizes the partial selection value $sv_r(m_k, M_i)$
4: $M_i = M_i \cup \{m_k\}$
5: $x_{ik} = 1$
6: $v_{ik} = V_{max}$
7: **end while**
8: **return** x_i

Figure 5. The greedy repair algorithm.

Procedure Elimination(x_i)

1: $M_i = \{m_j \in M | x_{ij} = 1\}$
2: $R_i = \{m_j \in M_i | pv_x(m_j, M_i) = 0\}$
3: **while** $R_i \neq \varnothing$ **do**
4: Choose a countermeasure $m_k \in R_i$ that maximizes the redundant value $rv(m_k, M_i)$
5: $M_i = M_i \setminus \{m_k\}$
6: $x_{ik} = 0$
7: $v_{ik} = -V_{max}$
8: $R_i = \{m_j \in M_i | pv_x(m_j, M_i) = 0\}$
9: **end while**
10: **return** x_i

Figure 6. The algorithm for eliminating redundant countermeasures from particle positions.

varying velocity clamping on the performance of the SwarmCAG-TVVC.

7.3 Greedy Repair Algorithm

The set of countermeasures represented by the particle position x_i may not be critical. In other words, it may not prevent all attack scenarios.

Let M_i be the set of countermeasures represented by x_i. For each countermeasure $m_j \notin M_i$, we define the *partial selection value* $sv_r(m_j, M_i)$ to be the ratio between the partial prevention value of the countermeasure and its weight.

$$sv_r(m_j, M_i) = pv_r(m_j, M_i)/w(m_j) \quad (22)$$

Figure 5 shows the greedy repair algorithm. At each step, the algorithm chooses a countermeasure $m_k \in M$ such that $m_k \notin M_i$ and it maximizes the partial selection value $sv_r(m_k, M_i)$. It then adds m_k to M_i and changes its corresponding element x_{ik} to 1. This is repeated until a critical set of countermeasures is obtained.

7.4 Redundant Countermeasure Elimination Algorithm

The critical set of countermeasures represented by the particle position x_i may contain redundant countermeasures, which must be eliminated. Figure 6 shows an algorithm used to eliminate redundant countermeasures of x_i. Let M_i be the critical set of countermeasures represented by x_i and R_i be the set of candidate redundant countermeasures of M_i. At each step, the algorithm chooses a candidate redundant countermeasure m_k from R_i that maximizes the redundant value $rv(m_k, M_i)$. It then removes m_k from M_i and changes its corresponding element x_{ik} to 0. This is repeated until a minimal critical set of countermeasures is obtained.

7.5 Local Search Heuristic

Many empirical studies show that global optimization algorithms lack exploitation abilities in later stages of the optimization process. This is also true for the basic PSO [24–26], however, it provides mechanisms to balance exploration and exploitation through proper settings of the inertia weight, acceleration coefficients, and velocity clamping. Many variations of the basic PSO have been proposed to address this problem [22]. Most of them first allow the algorithm to explore new regions, and when a good region is located, allow the algorithm to exploit the search space to refine solutions. This is a sequential approach to balancing exploration and exploitation [22].

Another approach is to embed a local optimizer in between the iterations of the global search heuristic. By doing this, exploration and exploitation occur in parallel [22]. Such hybrids of local and global search heuristics have been studied elaborately in the evolutionary computation paradigm [27, 28].

In SwarmCAG-TVVC, a local search heuristic is applied to the current position of each particle to improve them before their personal best positions are updated. The local search heuristic is based on the idea: given a particle position x_i and its corresponding critical set of countermeasures M_i, suppose there is a countermeasure $m_j \in M$ such that $m_j \notin M_i$ and $M_i \cup \{m_j\}$ contains at least a countermeasure other than m_j , say $m'_1, ..., m'_r$, with $r \geq 1$ that are redundant. The gain of the countermeasure m_j with respect to M_i is

$$g(m_j, M_i) = w_s(M_i) - w_s((M_i \setminus \{m'_1, ..., m'_r\}) \cup \{m_j\}) \quad (23)$$

We say $(M_i \setminus \{m'_1, ..., m'_r\}) \cup \{m_j\}$ is a better critical set of countermeasures than M_i if $g(m_j, M_i) > 0$. In this case, we call m_j a *candidate dominant* countermeasure.

Procedure LocalSearch(x_i)

1: $M_i = \{m_j \in M | x_{ij} = 1\}$
2: **while** improvement is possible **do**
3: Choose a countermeasure $m_k \in M$ such that $m_k \notin M_i$ and $g(m_k, M_i) > 0$
4: $M_i = M_i \cup \{m_k\}$
5: $x_{ik} = 1$
6: $v_{ik} = V_{max}$
7: Apply the greedy elimination algorithm to x_i
8: **end while**
9: **return** x_i

Figure 7. The local search heuristic.

As shown in Figure 7, the local search heuristic first chooses a candidate dominant countermeasure m_k and changes its corresponding element x_{ik} to 1. It then eliminates the redundant countermeasures of the new position using the algorithm already presented in Section 7.4 for elimination of redundant countermeasures. This process is repeated until no further improvement is possible.

8 Experiments

In order to evaluate the performance of SwarmCAG-TVVC, we performed our experiments over a sample cost-sensitive attack graph and several randomly generated large-scale cost-sensitive attack graphs. We also performed experiments to analyze the effect of different settings of parameters on the performance of the SwarmCAG-TVVC.

8.1 Sample Cost-Sensitive Attack Graph

Consider the network shown in Figure 8. There are three target hosts called *RedHat*, *Windows* and *Fedora* on an internal network, and a host called *PublicServer* on an isolated demilitarized zone (DMZ) network. One firewall separates the internal network from the DMZ and another firewall separates the DMZ from the rest of the Internet.

A number of services are running on each of the hosts of *RedHat*, *Windows*, *Fedora*, and *PublicServer*. Also, each of the above hosts has a number of vulnerabilities. Vulnerability scanning tools such as Nessus [23] can be used to find the vulnerabilities of each host. Different types of services and vulnerabilities available on the hosts are introduced in Table 1.

The *RedHat* host on the internal network is running FTP, SSH, and RSH services. The *Fedora* host is running several services: LICQ chat software, Squid web proxy, FTP, RSH, and MySQL database service. The *PublicServer* host on the DMZ network is running IIS and Exchange services.

Table 1. Types of services and vulnerabilities running on the hosts of the sample network.

iis_bof(h)	IIS web server has buffer overflow vulnerability on host h
exchange_ivv(h)	Exchange mail server has input validation vulnerability on host h
squid_conf(h)	Squid web proxy is misconfigured on host h
licq_ivv(h)	LICQ client has input validation vulnerability on host h
ssh_bof(h)	SSH server has buffer overflow vulnerability on host h
scripting(h)	HTML scripting is enabled on host h
ftp(h)	FTP service is running on host h
wdir(h)	FTP home directory is writable on host h
fshell(h)	FTP user has executable shell on host h
xterm_bof(h)	*xterm* program has buffer overflow vulnerability on host h
at_bof(h)	*at* program has buffer overflow vulnerability on host h
mysql(h)	MySQL database service is running on host h

The connectivity information among the network hosts is shown in Table 2. In this Table, each entry corresponds to a pair of (h_s, h_t) in which h_s is the source host and h_t is the target host. Every entry has six boolean values. These values are 'T' if host h_s can connect to host h_t on the ports of *http*, *licq*, *ftp*, *ssh*, *smtp*, and *rsh*, respectively.

The intruder launches his attack starting from a single host, *Intruder*, which lies on the outside network. His goal is to disrupt the MySQL database service on the host *Fedora*. To achieve this goal, the intruder should gain the *root* privilege on this host.

There are *wdir*, *fshell*, and *ssh_bof* vulnerabilities on the *RedHat* host, scripting vulnerability on the *Windows* host, *wdir*, *fshell*, *squid_conf*, and *licq_ivv* vulnerabilities on the *Fedora* host, and *iis_bof* and *exchange_ivv* on the *PublicServer* host. Also, *at* and *xterm* programs on the *RedHat* and *Fedora* are vulnerable to buffer overflow.

The intruder can use ten generic exploits. In Table 3, each generic exploit is represented by its preconditions and postconditions. The description of each generic exploit is given in Table 4. More information about each of the exploits is available in [29]. Before an exploit can be used, its preconditions must be met. Each exploit will increase the network vulnerability if it is successful.

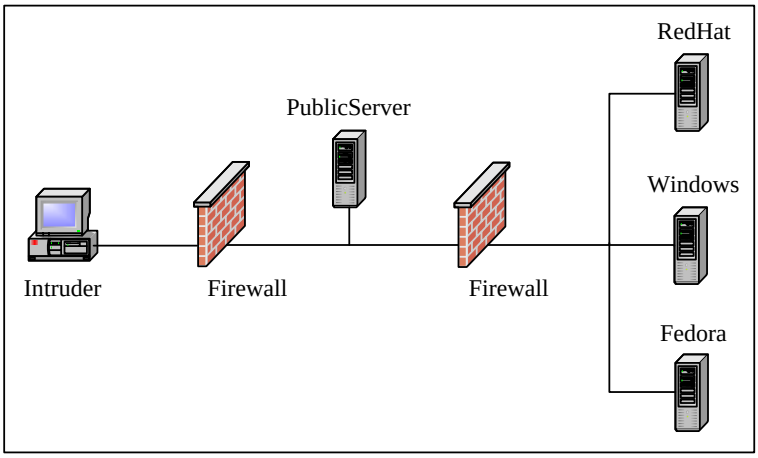

Figure 8. An example network.

Table 2. Network connectivity information.

Host	Intruder	PublicServer	RedHat	Windows	Fedora
Intruder	F,F,F,F,F,F	T,F,F,F,T,F	F,F,F,F,F,F	F,F,F,F,F,F	F,F,F,F,F,F
PublicServer	F,F,F,F,F,F	T,F,F,F,T,F	F,F,T,T,F,T	F,F,F,F,F,F	T,T,T,F,F,T
RedHat	F,F,F,F,F,F	T,F,F,F,T,F	F,F,T,T,F,T	F,F,F,F,F,F	T,T,T,F,F,T
Windows	F,F,F,F,F,F	T,F,F,F,T,F	F,F,T,T,F,T	F,F,F,F,F,F	T,T,T,F,F,T
Fedora	F,F,F,F,F,F	T,F,F,F,T,F	F,F,T,T,F,T	F,F,F,F,F,F	T,T,T,F,F,T

Among the ten generic exploits shown in Table 3, the first eight generic exploits require a pair of hosts and the last two generic exploits require only one host. Therefore, there are $8 * 5 * 4 + 2 * 4 = 168$ exploits in total, which the intruder can try. Each attack scenario for the above network consists of a subset of these exploits. For example, consider the following attack scenario:

(1) $iis\_r2r(Intruder, PublicServer)$

(2) $squid\_ps(PublicServer, Fedora)$

(3) $licq\_r2u(PublicServer, Fedora)$

(4) $xterm\_u2r(Fedora, Fedora)$

The intruder first launches the *iis_r2r* exploit to gain *root* privilege on the *PublicServer* host. Then, he uses the *PublicServer* host to launch a port scan via the vulnerable Squid web proxy running on the *Fedora* host. The scan discovers that it is possible to gain *user* privilege on the *Fedora* host with launching the *licq_r2u* exploit. After that, a simple local buffer overflow gives the intruder *root* privilege on the *Fedora* host.

Table 5 gives some examples of countermeasures for the exploits. The "*" in a field is a wildcard designator that matches every host. It is assumed that to prevent the *iis_r2r* and *exchange_r2u* exploits, which are launched against the *PublicServer* host, it is necessary for the security analyst to set up a host-based intrusion detection and prevention system on this host. Accordingly, the cost of implementing this countermeasure is considered *very high*. Preventing those exploits which are launched from the *PublicServer* host against other internal hosts (except for the *script_r2u* exploits) is simply possible by changing the configuration of the second firewall, so the cost of implementing the countermeasures for these exploits is considered *low*. Furthermore, to prevent the *squid_ps*, *licq_r2u*, *script_r2u*, *ssh_r2r*, *ftp_rhosts*, *xterm_u2r* and *at_u2r* exploits, which are launched against the *RedHat*, *Windows* and *Fedora* hosts, the security analyst can patch the *squid_conf*, *licq_ivv*, *ssh_bof*, *scripting*, *ftp_vul*, *xterm_bof* and *at_bof* vulnerabilities on these hosts, respectively. The cost of implementing these countermeasures is considered *medium*.

We assign weights 10, 30, 50, and 100 to the countermeasures with *low*, *medium*, *high* and *very high* costs, respectively. The above weights assigned to the countermeasures can be adjusted according to the security analyst's experience.

The cost-sensitive attack graph for the above network consists of 164 attack scenarios. Each attack

Table 3. Exploit templates.

Exploit	Preconditions	Postconditions
$iis\_r2r(h_s, h_t)$	$iis\_bof(h_t)$ $C(h_s, h_t, http)$ $plvl(h_s) \geq$ user $plvl(h_t) <$ root	$\neg iis(h_t)$ $plvl(h_t) :=$ root
$exchange\_r2u(h_s, h_t)$	$exchange\_ivv(h_t)$ $C(h_s, h_t, smtp)$ $plvl(h_s) \geq$ user $plvl(h_t) =$ none	$plvl(h_t) :=$ user
$squid\_ps(h_s, h_t)$	$squid\_conf(h_t)$ $\neg scan$ $C(h_s, h_t, http)$ $plvl(h_s) \geq$ user	$scan$
$licq\_r2u(h_s, h_t)$	$licq\_ivv(h_t)$ $scan$ $C(h_s, h_t, licq)$ $plvl(h_s) \geq$ user $plvl(h_t) =$ none	$plvl(h_t) :=$ user
$script\_r2u(h_s, h_t)$	$scripting(h_t)$ $C(h_t, h_s, http)$ $plvl(h_s) \geq$ user $plvl(h_t) =$ none	$plvl(h_t) :=$ user
$ssh\_r2r(h_s, h_t)$	$ssh\_bof(h_t)$ $C(h_s, h_t, ssh)$ $plvl(h_s) \geq$ user $plvl(h_t) <$ root	$\neg ssh(h_t)$ $plvl(h_t) :=$ root
$ftp\_rhosts(h_s, h_t)$	$ftp(h_t)$ $wdir(h_t)$ $fshell(h_t)$ $\neg T(h_t, h_s)$ $C(h_s, h_t, ftp)$ $plvl(h_s) \geq$ user	$T(h_t, h_s)$
$rsh\_r2u(h_s, h_t)$	$T(h_t, h_s)$ $C(h_s, h_t, rsh)$ $plvl(h_s) \geq$ user $plvl(h_t) =$ none	$plvl(h_t) :=$ user
$xterm\_u2r(h_t, h_t)$	$xterm\_bof(h_t)$ $plvl(h_t) =$ user	$plvl(h_t) :=$ root
$at\_u2r(h_t, h_t)$	$at\_bof(h_t)$ $plvl(h_t) =$ user	$plvl(h_t) :=$ root

Table 4. Description of generic exploits.

Exploit	Description
iis_r2r	Buffer overflow vulnerability in the IIS web server allows remote intruders to gain *root* shell on the target network host
exchange_r2u	The OLE component in the Microsoft Exchange mail server does not properly validate the lengths of messages for certain OLE data, which allows remote intruders to execute arbitrary code
squid_ps	The intruder can use a misconfigured Squid web proxy to conduct unauthorized activities such as port scanning
licq_r2u	The intruder can send a specially crafted URL to the LICQ client to execute arbitrary commands on the target network host
script_r2u	Microsoft Internet Explorer allows remote intruders to execute arbitrary code via malformed Content-Type and Content-Disposition header fields that cause the application for the spoofed file type to pass the file back to the operating system for handling rather than raise an error message
ssh_r2r	Buffer overflow vulnerability in the SSH server allows remote intruders to gain *root* shell on the target network host
ftp_rhosts	Using FTP vulnerability, the intruder creates a .rhosts file in the FTP home directory, creating a remote login trust relationship between his network host and the target network host
rsh_r2u	Using an existing remote login trust relationship between two hosts, the intruder logs in from one machine to another, getting a *user* shell without supplying a password
xterm_u2r	Buffer overflow vulnerability in the *xterm* program allows local users to gain *root* shell on the target network host
at_u2r	Buffer overflow vulnerability in the *at* program allows local users to gain *root* shell on the target network host

scenario consists of between 4 to 9 exploits

Experimental Results

We applied GreedyCAG, presented in Section 6, for minimization analysis of the above cost-sensitive attack graph and found the following critical set of countermeasures:

$$\begin{aligned} M_c = \{&block\_licq(PublicServer, Fedora),\\ &patch\_scripting(Windows),\\ &block\_ftp(PublicServer, RedHat),\\ &block\_ftp(PublicServer, Fedora),\\ &block\_ssh(PublicServer, RedHat)\} \end{aligned}$$

The weight of the above critical set of countermeasures is 70. By implementing these countermeasures the intruder cannot reach his goal using any attack scenario.

We also applied SwarmCAG-TVVC, presented in Section 7, for minimization analysis of the above cost-sensitive attack graph and found the following critical set of countermeasures:

$$\begin{aligned} M_c = \{&patch\_licq\_ivv(Fedora),\\ &patch\_ftp\_vul(Fedora)\} \end{aligned}$$

The weight of the above critical set of countermeasures is 60. It should be mentioned that the minimum weight of the critical set of countermeasures for the above cost-sensitive attack graph is also 60, so the critical

set of countermeasures found by SwarmCAG-TVVC has the minimum weight.

The above experiment shows SwarmCAG-TVVC can find a critical set of countermeasures with less weight.

In the experiments of SwarmCAG-TVVC, the parameters c_1 and c_2 were set to 2, which are values commonly used in the binary PSO literature. The swarm size was set to $m = 5$ and the maximum number of iterations was set to $t_{max} = 50$. The initial and final velocity clamping were set to $V_{max}(0) = 2$ and $V_{max}(t_{max}) = 4$. The probability of the local search was set to $P_l = 0.90$ and the starting time of the time-varying velocity clamping was set to $t_{tvvc} = 25$.

8.2 Large-Scale Cost-Sensitive Attack Graphs

Several factors can make attack graphs very large so that finding a minimum critical set of countermeasures becomes more difficult. An obvious factor is the size of the network under analysis. Our society has become increasingly dependent on networked computers and the trend towards larger networks will continue. For example, there are enterprises today consisting of tens of thousands of hosts. Also, less secure networks clearly have larger cost-sensitive attack graphs. Each network host might have several exploitable vulnerabilities.

When considered across an enterprise, especially given global internet connectivity, attack graphs become potentially large [30].

In order to further evaluate the performance of SwarmCAG-TVVC, we randomly generated 14 large-scale cost-sensitive attack graphs, denoted by WAG_1, WAG_2, ..., WAG_{14}. For each cost-sensitive attack graph, we considered different values for the cardinalities of E, M, and S, where E is the set of exploits, M is the set of countermeasures, and S is the set of attack scenarios represented by the cost-sensitive attack graph. With each exploit in E, between 1 to 3 countermeasures in M were associated. Each attack scenario in S consisted of between 3 to 9 exploits in E.

We randomly assigned 10 different weights between 1 to 100 to the countermeasures in M.

Table 6 shows the cardinalities of the sets of exploits, countermeasures, and attack scenarios for each generated large-scale cost-sensitive attack graph. It also shows the average degree and the total weight of the countermeasures.

Experimental Results

We first applied GreedyCAG, presented in Section 6, and SwarmCAG-TVVC, presented in Section 7, for minimization analysis of the large-scale cost-sensitive attack graphs of Table 6. We then applied GenNAG [2] for minimization analysis of the above large-scale cost-sensitive attack graphs. It should be mentioned that GenNAG is a genetic algorithm, presented in [2], for minimization analysis of simple attack graphs. For the sake of comparison, we slightly modified GenNAG, so that it can be used for minimization analysis of cost-sensitive attack graphs.

We performed 10 runs of SwarmCAG-TVVC and 10 runs of GenNAG with different random seeds and reported the best, the average, and the standard deviation of the weights of critical sets of countermeasures obtained from these 10 runs. As shown in Table 7, SwarmCAG-TVVC outperforms GreedyCAG and GenNAG.

In the experiments of SwarmCAG-TVVC, the parameters c_1 and c_2 were set to 2, which are values commonly used in the binary PSO literature. The swarm size was set to $m = 20$. The initial and final velocity clamping were set to $V_{max}(0) = 2$ and $V_{max}(t_{max}) = 4$. The probability of the local search was set to $P_l = 0.90$ and the starting time of the time-varying velocity clamping was set to $t_{tvvc} = 50$. In the experiments of GenNAG, the parameter settings were similar to the parameter settings proposed in [2]. The maximum number of iterations was set to $t_{max} = 200$ for all the above algorithms.

On average, the weight of a critical set of countermeasures found by SwarmCAG-TVVC is 6.15 percent less than the weight of a critical set of countermeasures found by GreedyCAG.

Figures 9 to 12 show the progress of the average total weight of countermeasures corresponding to the global best position of SwarmCAG and the best chromosome of GenNAG in the experiments for minimization analysis of WAG_4, WAG_7, WAG_{13} and WAG_{14}, respectively. As the figures show, SwarmCAG-TVVC performs better than GenNAG in terms of convergence speed and accuracy.

8.3 Algorithm Parameters

We performed experiments to analyze the effect of different settings of parameters on the performance of SwarmCAG-TVVC.

TVVC

The effect of using the time-varying velocity clamping on the performance of the proposed algorithm is analyzed by comparing the results of running the algorithm with and without the TVVC. We call the proposed algorithm without the TVVC as SwarmCAG. It uses a fixed velocity clamping. We applied SwarmCAG for minimization analysis of the large-scale cost-

Table 5. Examples of countermeasures.

Exploit	**Source**	**Target**	**Countermeasure**	**Cost**
iis_r2r	*	*PublicServer*	*idps*(*PublicServer*)	*very high*
exchange_r2u	*	*PublicServer*	*idps*(*PublicServer*)	*very high*
squid_ps	*	*Fedora*	*patch_squid_conf*(*Fedora*)	*medium*
squid_ps	*PublicServer*	*Fedora*	*block_http*(*PublicServer*, *Fedora*)	*low*
licq_r2u	*	*Fedora*	*patch_licq_ivv*(*Fedora*)	*medium*
licq_r2u	*PublicServer*	*Fedora*	*block_licq*(*PublicServer*, *Fedora*)	*low*
script_r2u	*	*Windows*	*patch_scripting*(*Windows*)	*medium*
ssh_r2r	*	*RedHat*	*patch_ssh_bof*(*RedHat*)	*medium*
ssh_r2r	*PublicServer*	*RedHat*	*block_ssh*(*PublicServer*, *RedHat*)	*low*
ftp_rhosts	*	*Fedora*	*patch_ftp_vul*(*Fedora*)	*medium*
ftp_rhosts	*PublicServer*	*Fedora*	*block_ftp*(*PublicServer*, *Fedora*)	*low*
rsh_r2u	*PublicServer*	*RedHat*	*block_rsh*(*PublicServer*, *RedHat*)	*low*
xterm_u2r	*Fedora*	*Fedora*	*patch_xterm_bof*(*Fedora*)	*medium*
at_u2r	*Fedora*	*Fedora*	*patch_at_bof*(*Fedora*)	*medium*

Table 6. Large-scale cost-sensitive attack graphs.

Cost-sensitive Attack Graph	Cardinality of the Set of Exploits	Cardinality of the Set of Countermeasures	Cardinality of the Set of Attack Scenarios	Average Degree of Countermeasures	Total Weight of Countermeasures
CAG_1	200	177	2000	2.15	8837
CAG_2	200	250	2000	1.58	12579
CAG_3	400	338	4000	2.29	15803
CAG_4	400	516	4000	1.58	24738
CAG_5	400	346	6000	2.32	17513
CAG_6	400	507	6000	1.53	26357
CAG_7	600	516	6000	2.28	25785
CAG_8	600	767	6000	1.53	38982
CAG_9	600	513	8000	2.26	24618
CAG_{10}	600	765	8000	1.58	37509
CAG_{11}	800	693	8000	2.35	34279
CAG_{12}	800	1009	8000	1.60	51047
CAG_{13}	1000	873	10000	2.30	44708
CAG_{14}	1000	1264	10000	1.57	64776

Table 7. The weights of critical sets of countermeasures found by SwarmCAG-TVVC, GenNAG, and GreedyCAG.

Cost-sensitive Attack Graph	SwarmCAG-TVVC			GenNAG [2]			GreedyCAG
	Best	Average	Std. Dev.	Best	Average	Std. Dev.	
CAG_1	1003	1003.0	0.00	1003	1003.0	0.00	1033
CAG_2	1488	1494.6	10.63	1488	1495.0	10.42	1621
CAG_3	1867	1872.9	12.55	1867	1876.2	14.94	2051
CAG_4	2433	2443.3	5.38	2441	2458.8	13.72	2670
CAG_5	2334	2338.9	9.43	2334	2341.8	5.92	2533
CAG_6	3295	3295.0	0.00	3295	3337.6	36.06	3514
CAG_7	3175	3187.2	13.28	3175	3199.2	9.07	3375
CAG_8	4066	4085.8	16.14	4076	4130.8	34.85	4317
CAG_9	3537	3554.0	12.16	3529	3563.4	28.45	3666
CAG_{10}	4308	4336.7	13.61	4308	4339.9	21.43	4738
CAG_{11}	3779	3789.5	12.15	3771	3798.2	30.46	3955
CAG_{12}	4951	4983.3	21.56	4974	5000.2	22.98	5236
CAG_{13}	5469	5514.7	16.11	5519	5554.5	42.23	5872
CAG_{14}	6968	7042.2	50.15	7017	7146	63.44	7553

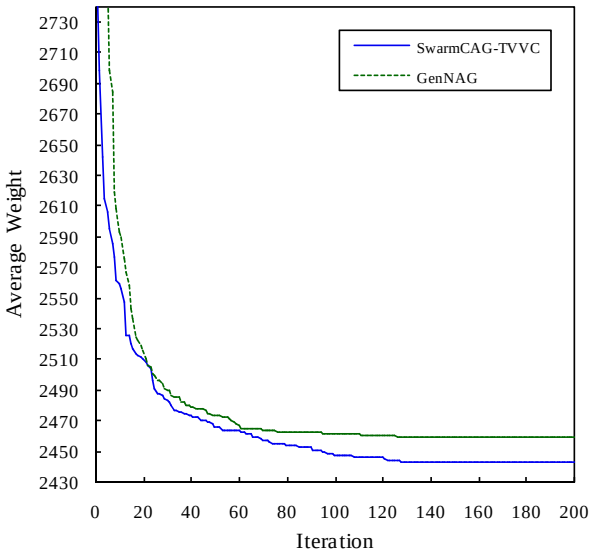

Figure 9. Comparison of the performance of SwarmCAG and GenNAG for minimization analysis of CAG_4.

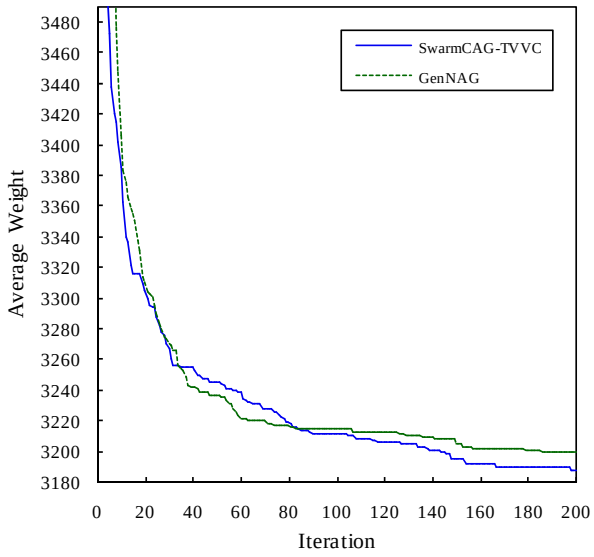

Figure 10. Comparison of the performance of SwarmCAG and GenNAG for minimization analysis of CAG_7.

sensitive attack graphs of Table 6. In the experiments of SwarmCAG, the parameter settings were similar to the parameter settings of SwarmCAG-TVVC in Section 8.2, but with this difference that the velocity clamping V_{max} was set to a fixed value. Table 8 shows the results of 10 runs of the SwarmCAG for different settings of V_{max}.

Figures 13 and 14 show the progress of the average total weight of countermeasures corresponding to the global best position of SwarmCAG and SwarmCAG-TVVC in the experiments for minimization analysis of WAG_7 and WAG_{13}, respectively. The total weight of countermeasures corresponding to the global best position is expected to be as small as possible. As the figures show, SwarmCAG-TVVC significantly performs better than the SwarmCAG and finds a critical set of countermeasures with less weight.

Diversity is an important measure that may be used

Table 8. The average weights of critical sets of countermeasures found by the SwarmCAG.

Cost-sensitive Attack Graph	SwarmCAG ($V_{max} = 1.5$)	SwarmCAG ($V_{max} = 2$)	SwarmCAG ($V_{max} = 4$)
CAG_1	1003.0	1003.9	1006.4
CAG_2	1494.6	1502.3	1507.8
CAG_3	1887.2	1879.5	1887.5
CAG_4	2480.3	2453.1	2452.9
CAG_5	2339.0	2338.0	2345.0
CAG_6	3303.8	3295.0	3295.0
CAG_7	3266.4	3201.8	3211.3
CAG_8	4206.1	4117.2	4090.5
CAG_9	3588.1	3560.7	3562.6
CAG_{10}	4511.7	4368.4	4342.3
CAG_{11}	3911.0	3818.1	3791.5
CAG_{12}	5190.7	5015.6	4986.2
CAG_{13}	5703.1	5531.1	5523.9
CAG_{14}	7455.5	7209.5	7053.6

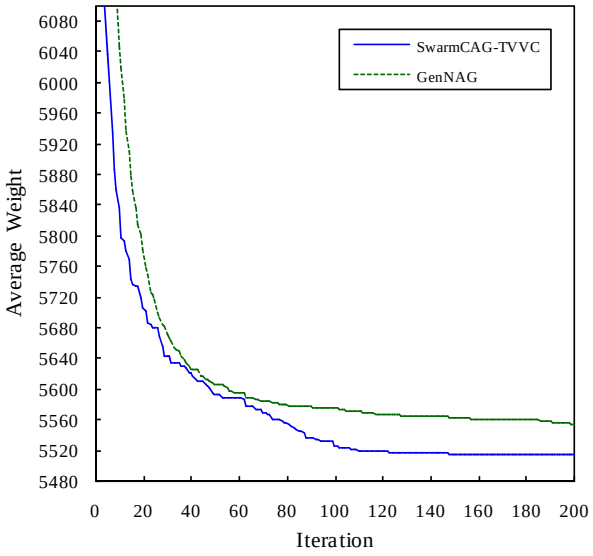

Figure 11. Comparison of the performance of SwarmCAG and GenNAG for minimization analysis of CAG_{13}.

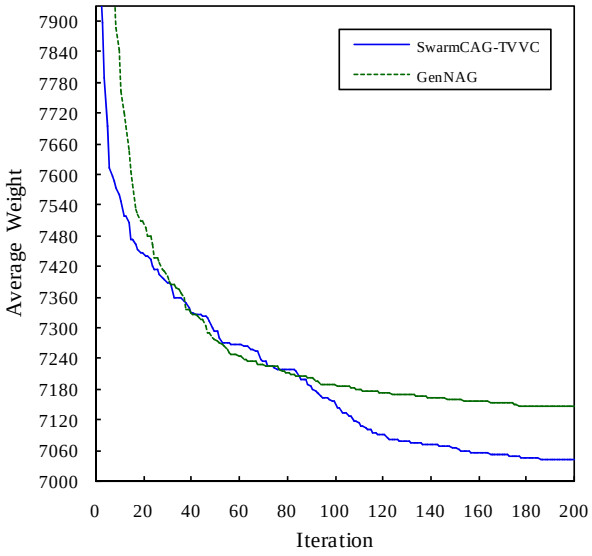

Figure 12. Comparison of the performance of SwarmCAG and GenNAG for minimization analysis of CAG_{14}.

to describe the amount of exploration a PSO algorithm still performs and to detect stagnation situations. Large diversity implies that a large area of the search space can be explored. We used the diversity measure in [17] to compare the amount of exploration the SwarmCAG and SwarmCAG-TVVC perform.

Figures 15 and 16 show the progress of the average diversity of SwarmCAG and SwarmCAG-TVVC in the experiments for minimization analysis of WAG_7 and WAG_{13}, respectively. Large average diversity implies that an algorithm can explore a large area of the search space.

As the figures 15 and 16 show, SwarmCAG with $V_{max} = 1.5$ or $V_{max} = 2$ allows particles to explore the search space, but they are not able to quickly converge on a good solution. SwarmCAG with $V_{max} = 1.5$ preserves a very large diversity during the running of the algorithm, hence it performs similar to a random search. SwarmCAG with $V_{max} = 4$ allows particles to exploit the search space, but they are not able

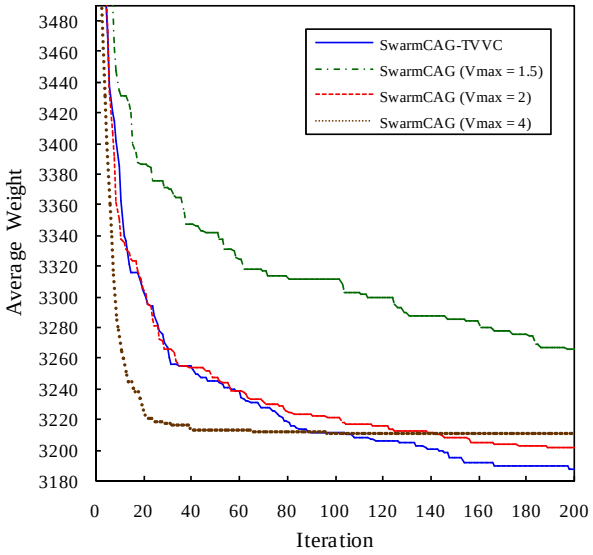

Figure 13. Effect of the time-varying velocity clamping on the performance of SwarmCAG-TVVC in the experiments for minimization analysis of CAG_7.

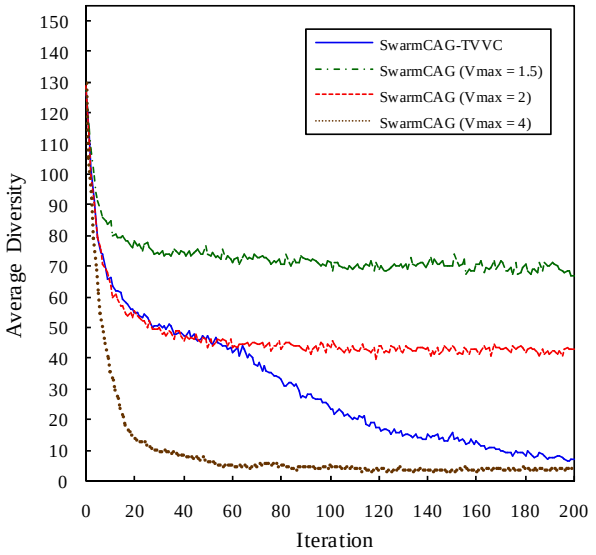

Figure 15. Comparison of the average diversity of SwarmCAG-TVVC and SwarmCAG in the experiments for minimization analysis of CAG_7.

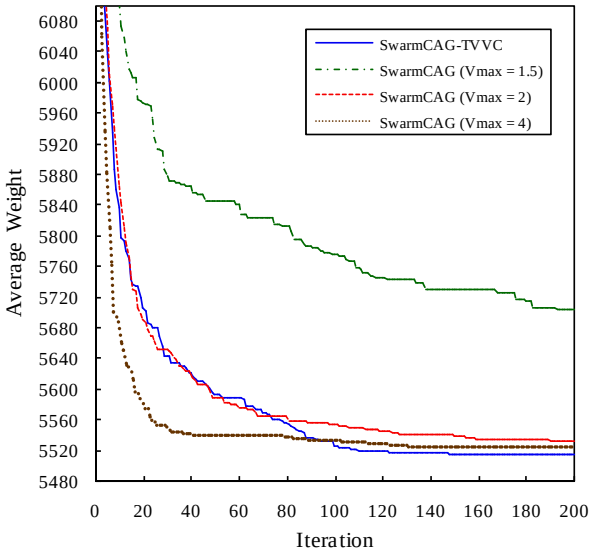

Figure 14. Effect of the time-varying velocity clamping on the performance of SwarmCAG-TVVC in the experiments for minimization analysis of CAG_{13}.

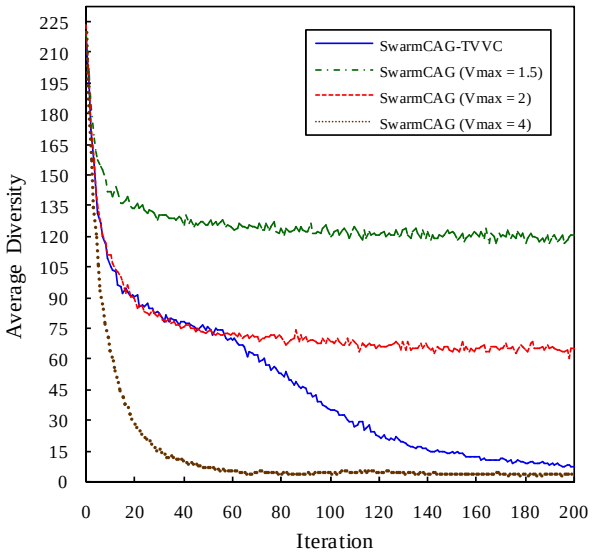

Figure 16. Comparison of the average diversity of SwarmCAG-TVVC and SwarmCAG in the experiments for minimization analysis of CAG_{13}.

to effectively explore the search space. In contrast, SwarmCAG-TVVC allows particles to explore the search space in the initial iterations, and afterwards, it allows particles to exploit the search space to refine solutions. In fact, SwarmCAG-TVVC balances the exploration and exploitation through proper setting of velocity clamping.

Swarm Size

To analyze the effect of the swarm size on the performance of SwarmCAG-TVVC, the algorithm was run with the parameter settings from Section 8.2 but this time with the swarm size set to 2 and 10, respectively.

Figures 17 and 18 show the effect of the swarm size on the performance of SwarmCAG-TVVC in the experiments for minimization analysis of WAG_4 and WAG_7, respectively. As the figures show, when using a very small number of particles, SwarmCAG-TVVC shows a poor performance. This is because the fewer the number of particles, the less the exploration ability of the algorithm, and consequently the less information about the search space is available to all particles.

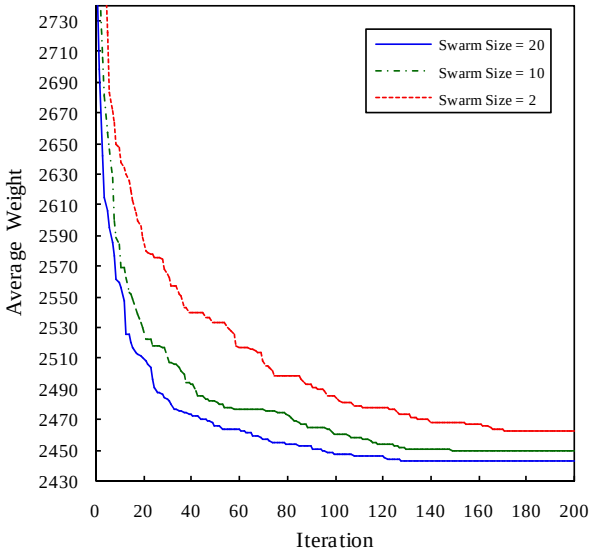

Figure 17. Effect of the swarm size on the performance of SwarmCAG-TVVC in the experiments for minimization analysis of CAG_4.

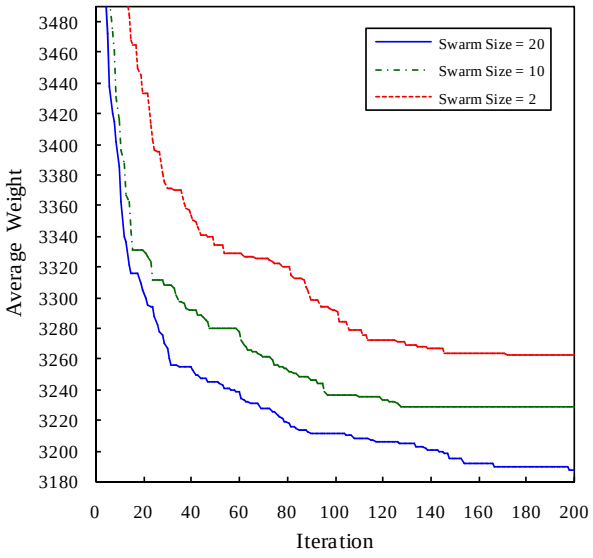

Figure 18. Effect of the swarm size on the performance of SwarmCAG-TVVC in the experiments for minimization analysis of CAG_7.

Local Search Heuristic

The effect of using the local search heuristic on the performance of SwarmCAG-TVVC is analyzed by comparing the results of running the algorithm with and without the local search heuristic. Figures 19 and 20 show the effect of the local search heuristic on the performance of SwarmCAG-TVVC in the experiments for minimization analysis of WAG_7 and WAG_{13}, respectively. As the figures show, during the running of the algorithm, SwarmCAG-TVVC significantly performs better than SwarmCAG-TVVC without the local search heuristic.

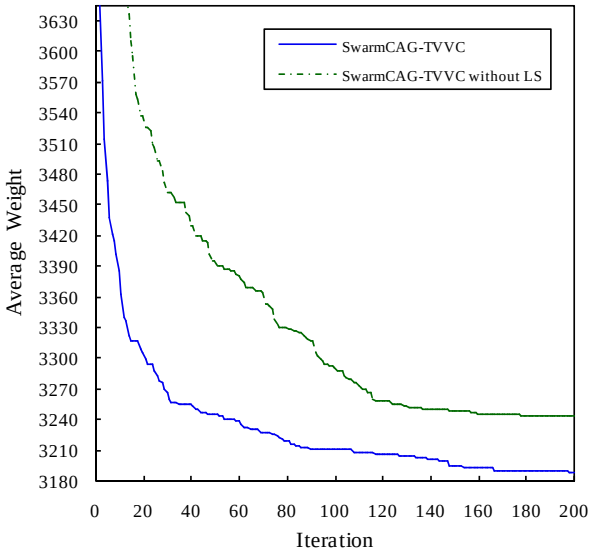

Figure 19. EEffect of the local search heuristic on the performance of SwarmCAG-TVVC in the experiments for minimization analysis of CAG_7.

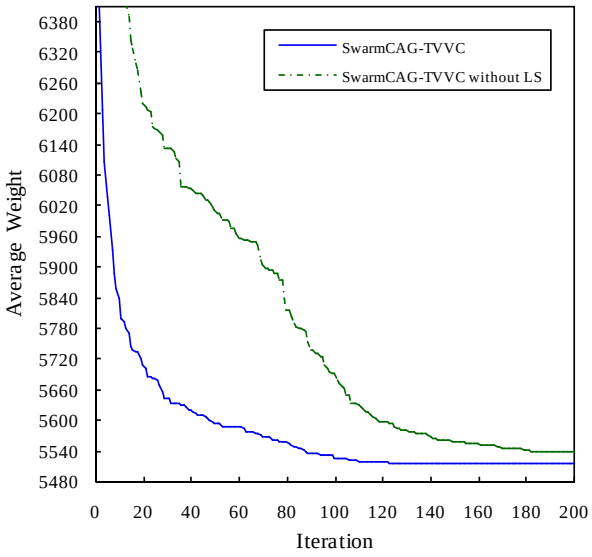

Figure 20. Effect of the local search heuristic on the performance of SwarmCAG-TVVC in the experiments for minimization analysis of CAG_{13}.

9 Discussion and Conclusions

Each attack scenario is a sequence of exploits launched by an intruder for a particular goal such as access to a database or service disruption. The collection of possible attack scenarios in a computer network can be represented by a directed graph, called attack graph. While it is currently possible to generate very large and complex attack graphs, relatively little work has been done to analyze them.

Sheyner *et al.* [6] and Jha *et al.* [15, 16] presented an

approximation algorithm for minimization analysis of simple attack graphs. The aim is to find a minimum critical set of exploits so that by preventing them no attack scenario is possible. Abadi and Jalili [2] presented a genetic algorithm GenNAG for minimization analysis of simple attack graphs. In the above algorithms, the cost of preventing exploits is considered to be the same while in the real world, the cost of preventing an exploit could be different from the cost of preventing another exploit. For example, in order to prevent an exploit, the security analyst may change the firewall configuration, patch the vulnerability that made this exploit possible, or deploy an intrusion detection and prevention system. Noel *et al.* [11] and Wang *et al.* [12] proposed a solution to automate the task of hardening a network against attack scenarios. They presented a procedure for choosing a minimum cost solution, but it has an unavoidable exponential worst-case complexity [12] and cannot be employed in a large enterprise network.

In this paper, we considered cost-sensitive attack graphs for network vulnerability analysis. In these attack graphs, a weight is assigned to each countermeasure to represent the cost of its implementation. There may be multiple countermeasures with different weights for preventing a single exploit. Also, a single countermeasure may prevent multiple exploits. The aim of minimization analysis of a cost-sensitive attack graph is to find a critical set of countermeasures with minimum weight whose implementation causes the initial nodes and the goal nodes of the graph to be completely disconnected. This problem is in fact a constrained optimization problem in which the objective is to find a solution with minimum weight and the constraint is that the solution must be critical.

We presented a binary PSO algorithm with a time-varying velocity clamping, called SwarmCAG-TVVC, for minimization analysis of cost-sensitive attack graphs. SwarmCAG-TVVC extends the binary PSO for minimization analysis of cost-sensitive attack graphs by incorporating a time-varying velocity clamping, a greedy repair algorithm, a redundant countermeasure elimination algorithm, and a local search heuristic. The greedy repair algorithm is used to convert the constrained optimization problem into an unconstrained one. The local search heuristic is used to improve the overall performance of the algorithm.

We reported the results of applying SwarmCAG-TVVC for minimization analysis of several large-scale cost-sensitive attack graphs. We also applied GreedyCAG and GenNAG [2] for minimization analysis of the above large-scale cost-sensitive attack graphs. The aim of presenting GreedyCAG was to provide the possibility of comparing the performance of SwarmCAG-TVVC with an approximation algorithm for minimization analysis of cost-sensitive attack graphs. GenNAG [2] is a genetic algorithm for minimization analysis of simple attack graphs. For the sake of comparison, we slightly modified GenNAG, so that it can be used for minimization analysis of cost-sensitive attack graphs.

On average, the weight of critical sets of exploits found by SwarmCAG-TVVC was 6.15 percent less than the weight of critical sets of exploits found by GreedyCAG. Also, SwarmCAG-TVVC performed better than GenNAG [2] in terms of convergence speed and accuracy.

We performed experiments to analyze the effect of TVVC, swarm size, and local search heuristic on the performance of SwarmCAG-TVVC. The results of experiments showed that SwarmCAG-TVVC can effectively balance between the exploration and exploitation of the search space through proper setting of velocity clamping. Also, SwarmCAG-TVVC significantly performs better than SwarmCAG-TVVC without the local search heuristic.

Acknowledgements

This work was supported in part by the Iran Telecommunication Research Center (ITRC).

References

[1] 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 Challenges*, pages 247 – 266. Springer, New York, NY, USA, 2005.

[2] M. Abadi and S. Jalili. Minimization Analysis of Network Attack Graphs Using Genetic Algorithms. *International Journal of Computers and Their Applications (IJCA)*, 15(4):263 – 273, 2008.

[3] J. Kennedy and R. C. Eberhart. Particle Swarm Optimization. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 1942 – 1948, Perth, Australia, 1995.

[4] C. Phillips and L. P. Swiler. A Graph-based System for Network-Vulnerability Analysis. In *Proceedings of the New Security Paradigms Workshop*, pages 71 – 79, Charlottesville, VA, USA, 1998.

[5] J. M. Wing. Attack Graph Generation and Analysis. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, page 14, Taipei, Taiwan, 2006.

[6] O. Sheyner, J. W. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 273 – 284, Berkeley, CA, USA, 2002.

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

[8] P. Ammann, D. Wijesekera, Kaushik, and S. Scalable, Graph-based Network Vulnerability Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217 – 224, Washington, DC, USA, 2002.

[9] S. Noel, M. Jacobs, P. Kalapa, and S. Jajodia. Multiple Coordinated Views for Network Attack Graphs. In *Proceedings of the IEEE Workshop on Visualization for Computer Security (VizSEC 2005)*, pages 99 – 106, Minneapolis, MN, USA, 2005.

[10] V. Mehta, C. Bartzis, H. Zhu, E. M. Clarke, and J. M. Wing. Ranking Attack Graphs. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006)*, pages 127 – 144, Hamburg, Germany, 2006.

[11] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient Minimum-Cost Network Hardening via Exploit Dependency Graphs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 86 – 95, Las Vegas, NV, USA, 2003.

[12] L. Wang, S. Noel, and S. Jajodia. Minimum-Cost Network Hardening Uusing Attack Graphs. *Computer Communications*, 29(18):3812 – 3824, 2006.

[13] X. Ou, W. F. Boyer, and M. A. McQueen. A Scalable Approach to Attack Graph Generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 336 – 345, Alexandria, VA, USA, 2006.

[14] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A Logic-based Network Security Analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium*, page 8, Baltimore, MD, USA, 2005.

[15] S. Jha, O. Sheyner, and J. Wing. Minimization and Reliability Analysis of Attack Graphs. Technical report, CMU-CS-02-109, School of Computer Science, Carnegie Mellon University, 2002.

[16] S. Jha, O. Sheyner, and J. M. Wing. Two Formal Analyses of Attack Graphs. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 49 – 63, Cape Breton, Nova Scotia, Canada, 2002.

[17] M. Abadi and S. Jalili. Using Binary Particle Swarm Optimization for Minimization Analysis of Large-Scale Network Attack Graphs. *Journal of Scientia Iranica*, 15(6):605 – 619, 2008.

[18] J. Kennedy, R. C. Eberhart, and Y. Shi. *Swarm Intelligence*. Morgan Kaufmann, San Mateo, CA, USA, 2001.

[19] R. C. Eberhart, P. Simpson, and R. Dobbins. *Computational Intelligence PC Tools*. Academic Press Professional, San Diego, CA, USA, 1996.

[20] Y. Shi. Particle Swarm Optimization. *IEEE Connections*, 2(1):8 – 13, 2004.

[21] J. Kennedy and R. C. Eberhart. A Discrete Binary Version of the Particle Swarm Algorithm. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 4104 – 4109, Orlando, FL, USA, 1997.

[22] A. P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence*. John Wiley & Sons, Hoboken, NJ, USA, 2005.

[23] R. Deraison. Nessus Scanner. http://www.nessus.org.

[24] Y. Shi and R. C. Eberhart. Empirical Study of Particle Swarm Optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1945 – 1950, Washington, DC, USA, 1999.

[25] T. Hendtlass and M. Randall. A Survey of Ant Colony and Particle Swarm Meta-Heuristics and Their Application to Discrete Optimization Problems. In *Proceedings of the Inaugural Workshop on Artificial Life*, pages 15 – 25, Adelaide, Australia, 2001.

[26] D. Braendler and T. Hendtlass. The Suitability of Particle Swarm Optimisation for Training Neural Hardware. In *Proceedings of the 15th International Conference on Industrial and Engineering, Applications of Artificial Intelligence and Expert Systems*, pages 190 – 199, Cairns, Australia, 2002.

[27] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, Berlin, Germany, 2003.

[28] N. Krasnogor, A. Aragon, and J. Pacheco. Memetic Algorithms. In E. Alba and R. Mart, editors, *Metaheuristic Procedures for Training Neural Networks*, pages 225 – 248. Springer-Verlag, Berlin, Germany, 2006.

[29] NVD: National Vulnerability Database. http://nvd.nist.gov/.

[30] P. Ammann, J. Pamula, R. Ritchey, and J. Street. A Host-Based Approach to Network Attack Chaining Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC05)*, pages 72 – 84, Tucson, AZ, USA, 2005.

Mahdi Abadi received the B.Sc. and M.Sc. degrees in computer engineering from Ferdowsi University of Mashhad in 1998 and Tarbiat Modares University in 2001, respectively. He also received the Ph.D. degree from Tarbiat Modares University in 2008, where he worked on the network vulnerability analysis. Since 2009, he has been an assistant professor in the Department of Computer Engineering at Tarbiat Modares University. His main research interests are network security, intrusion detection and prevention, evolutionary algorithms, data mining, and formal verification.

Saeed Jalili received the M.Sc. degree in computer science from Sharif University of Technology in 1979 and the Ph.D. degree from Bradford University in 1991. Since 1992, he has been an assistant professor in the Department of Computer Engineering at Tarbiat Modares University. His main research interests are machine learning, intrusion detection and prevention, runtime verification, and security protocol verification.