Multi-objective Software Assignment for Active Cyber Defense

Chu Huang
School of Information Science and Technology
Pennsylvania State University
cuh171@psu.edu

Sencun Zhu
Department of Computer Science and Engineering
Pennsylvania State University
szhu@cse.psu.edu

Quanlong Guan
Network and Education Technology Center
Jinan University
gql@jnu.edu.cn

***Abstract*—Software diversity is a well-accepted security principle for active cyber defense against the spread of Internet worms and other malicious attacks. In recent years, various software assignment techniques have been designed to introduce heterogeneity into network hosts for the maximum network survivability. However, few work consider practical constraints involved in the software assignment process. To close such a gap, in this work we model the software assignment problem as a multi-objective optimization problem, which incorporates several real-world criteria simultaneously, including network survivability, system feasibility and usability. To solves this multi-objective problem efficiently, we propose an ant colony optimization (ACO) based algorithm, where colonies of artificial ants work collaboratively through both heuristic information and pheromone-mediated communication to iteratively search for better solutions. To validate the generalizability of the proposed method, we experiment our algorithm on various types of network topologies with different parameter settings. The results show that our model can be applied as an effective method for assigning software for multiple objectives. The experimental results also provide interesting insights for optimal software assignment.**

.

I. Introduction

The use of software diversity has been recognized as an effective defense against virus and network attacks. To break up software monoculture, many different software assignment approaches have been proposed to introduce diversity for creating a heterogeneous environment. By allocating different software packages over the network in an intelligent way, those methods are able to increase the difficulty for an attacker to construct a spreading malware that exploits common vulnerabilities in the software [1]. Most software assignment works only have the single objective of maximum network survivability [2] [3]. However, in real world applications, security is not the only decision factor.

The practical software assignment problem may have various concerns and criteria other than security, such as feasibility of the system, user's satisfaction, etc. The transition from single-objective to multi-objective software assignment introduces great challenges, as multiple objectives might be incoherent or conflicting with each other. The interaction of different objectives gives rise to a set of solutions with different trade-offs, known as *Pareto solutions*. The previously proposed methods on software assignment work well on finding the "best" answer, but they are inapplicable when no single best solution exists.

This paper copes with the software assignment problem from a practical point of view. Specifically, we propose three important objectives that should be considered while assigning software to computers within a network, including: 1) maximizing network survivability; 2) maximizing system feasibility; and 3) maximizing usability of software. All three objectives are represented with corresponding objective functions in our model. We further present an Ant Colony Optimization (ACO) based algorithm to optimize the multi-objective software assignment problem. Rather than finding the single optimal solution that is best with respect to all the objectives, we show that it is possible to identify a set of solutions that are considered equally optimal for our software assignment problem. The proposed algorithm constructs probabilistic decisions on software choices for each host in the network, and thus provides sufficient flexibility and options in order to meet different needs and preferences.

The contributions of our work are summarized below. First, this work explores the scope of software diversity and the need for practical guidelines on constructing diverse software environments. Specifically, we model three software assignment criteria, and to our knowledge, it is the first model that considers software assignment problem from both the system and user's perspectives. Second, we propose an ACO algorithm to solve the multi-objective software assignment problem, in which the assignment choices are made probabilistically, depending on a combination of the heuristic factor (i.e., local evaluation of the decision in terms of the objectives) and pheromone information (i.e., the learnt desirability of decisions). This method helps in designing schemes for achieving multiple objectives simultaneously and provides a guideline for assigning software with flexibility. Finally, we conduct a comprehensive set of numerical experiments and provide some useful insights.

Quanlong Guan is the contact author. He can be reached by phone at 86-0-13430214465 or via email at gql@jnu.edu.cn

II. Related Work

A. Software Diversity

Many studies have been proposed to break up software monoculture in networked computer systems. Forrest et al. [4] [5] applied notions from biologic systems to computer security and promoted the general philosophy of system security using diversity. They presented the potential risks of software uniformity and also claimed that the security and robustness of a system can be enhanced through the deliberate introduction of diversity. Caballero et al. [6] exploited the existing diversity in router technology to design a network topology that has a diverse routing infrastructure. They explored the impact of different levels of diversity by converting the problem into a graph coloring task, and showed that well designed topology actually increased the global robustness of the infrastructure. Another work that leveraged natural diversity of software packages is done by [1]. They investigated several algorithms to increase the global diversity in a network. They modeled the diversification of distributed machines as a graph coloring problem and compared different algorithms according to their tolerance to attacks. Another work on software assignment was proposed by [2]. They highlighted the similar idea and presented an efficient algorithm for assigning different versions of software programs that effectively restrained worm propagation. Gorbenko et al. [8] also presented an approach which utilized the natural diversity of the off-the-shelf software components in the cloud (such as operating system, web server, database management system and application server) in order to build an intrusion avoidance architecture. In [3], Huang et al. proposed a software assignment approach based on graph multi-coloring algorithm. They abstracted the network as a graph and software as colors, and assigned software in a way that isolates worm-like attacks within a small cluster consisting of machines that run the same vulnerable software. This method, however, only addresses the single objective case.

Another category of the diversity-inspired approaches were based on some forms of obfuscation/randomization. Such techniques are able to create diversity of executions for application programs, including diverse performances, diverse outputs, diverse memory locations, etc. One of the works on randomization was done by Cohen [9], in which the author suggested the use of code transformation to protect operating systems from attacks or viruses. Forrest et al. [4] also noticed the advantages of diversity and proposed the use of variance-enhancing techniques such as adding or deleting non-functional code into a program, as well as reordering the basic blocks of a program in order to introduce diversity into the applications. The concept of instruction set randomization has been proposed later [10] [11], which created a unique mapping between artificial and real CPU instructions. Bathkar et al. [12] [13] proposed an approach to mitigate memory error exploits based on three kinds of randomization transformation techniques including randomization of the base addresses and libraries memory regions, permutation of the order of variables and routines, and the randomization of gaps between objects.

B. Ant Colony Optimization (ACO)

The very first ACO algorithm is a meta-heuristic approach proposed in [14], which is inspired by ants foraging behavior. At the core of this behavior is the ants' communication by using a chemical substance, called pheromone, which allows them to cooperate in finding the shortest path between their nests and food source. This approach was originally developed to solve the Traveling Salesman Problem (TSP) [15]. Many studies have also applied ACO to solve NP-hard discrete and continuous optimization problems, such as flow shop scheduling problem [16], vehicle routing, quadratic assignment problems, and graph coloring [17]. It is one of the most recent techniques for approximate optimization. Many variants of the basic principle of ACO have been proposed in literatures. An improving ACO algorithm was presented by Max-Min Ant System (MMAS) [18]. In this approach, only the current best solution in each iteration can contribute to pheromone deposition. Therefore, poor-quality solution will not be carried forward to the next iteration.

III. Problem Formulation

In order to illustrate the software assignment problem more clearly, we abstract a network as an undirected graph and present the following model to formalize our idea.

A. System Model

Definition 1 (*Weighted Communication Graph*) A weighted communication graph (WCG) is a weighted undirected graph $G = (V, E, W)$, where $V = \{v_1, v_2, \ldots, v_n\}$ represents a set of machines in the network, $E = \{e_1, e_2, \ldots, e_m\}$ represents the communication links (e.g., in network layer or application layer) between two nodes, and the weight vector $W = \{w_1, w_2, \ldots, w_n\}$ denotes the size for each vertex, which equals to the number of software to be installed on each node. Example WCGs include data center server farms, intranet, enterprise social networks, wireless sensor networks of different network topologies.

We use a vector $S = \{s_1, s_2, \ldots, s_k\}$ to denote k unique pieces of software. The task of software assignment is to assign a set of software $x(v_i) \subseteq S$ to every node v_i of G, where $|x(v_i)| = w_i$. The number of software to be installed can be estimated according to machine's functionality.

A perfect software assignment should guarantee all adjacent nodes in a WCG to be installed with different software packages in order to control the propagation of attacks such as worms through the network structure. In practice, however, due to the limited number of available software and particular system functional constraints, *defective edges* are induced. A *defective edge* is a communication link between two nodes with common software package, whereas an *immune edge* connects two nodes with different software. An edge in the WCG is either *defective* or *immune*. A worm-like attack can spread from one node to another if there is a defective edge

in between. Based on the above definitions, we further define the concept of common vulnerability graph.

Definition 2 (*Common Vulnerability Graph*) A common vulnerability graph (CVG), $G_{s_i}(V_{s_i}, E_{s_i})$, is a subgraph of WCG, where all nodes $v \in V_{s_i}$ are installed with the same software (hence every $e \in E_{s_i}$ is a defective edge) while all the entry and exit edges of G_{s_i} are immune edges.

In practical cases, each type of software s_i forms more than one CVG. The size of the maximum CVG $n_{maxs_i} = |V_{s_i}|$ reflects the worst case scenario if the vulnerability resides in software s_i is successfully exploited.

B. Formulation of Objectives

Objective 1: Maximizing Survivability

The way we improve network survivability is by minimizing attack severity. Severity is defined in terms of the potential damage caused by a vulnerable software. In general, the severity of a software is related to two factors: 1) the maximum number of connected nodes installed with the same software; 2) the severity level of the vulnerability, and it can be formulated as below:

$$SV_{s_i} = n_{maxs_i} * sl_{s_i} \tag{1}$$

where n_{maxs_i} signifies the largest size of the connected machines that are all installed with software s_i, and sl_{s_i} indicates the severity level of the vulnerability residing in software s_i. To minimize the severity, we need to control the highest potential risk by a single software vulnerability. The objective function can be formulated as:

$$\begin{aligned} Minimize \quad SV = & max\,[n_{maxs_1} * sl_{s_1}, n_{maxs_2} * sl_{s_2}, \cdots, \\ & n_{maxs_k} * sl_{s_k}] \end{aligned} \tag{2}$$

Objective 2: Maximizing Feasibility

Feasibility is defined as the degree to which the software assignment satisfies a set of functionality rules. Given that some constraints are more critical than the others, we propose two different types: hard constraint and soft constraint. A hard constraint is one that must be satisfied at all times. One example of hard constraints is the requirement for software installation—software that are incompatible with each other should not be installed together, or otherwise they will make the system easy to crash. A soft constraint is a rule that it is desirable, but not mandatory, to satisfy. Compared with the hard constraints, the soft ones can be relaxed when necessary.

Since the hard constraints are mandatory requirements and can never be violated, in order to maximize the system feasibility, we need to minimize the total violation of soft constraints as much as possible. Here we introduce a measurement called total *penalty score* P to quantify the overall cost of soft constraint violations. A good assigning solution should well accommodate the soft constraints, thus, our second objective is to minimize the total penalty score P, as listed below:

$$Minimize \qquad P = \sum_{i=1}^{n} \sum_{k=1}^{r} p_{ik} * d_{ik} \tag{3}$$

where p_{ik} denotes the penalty score if the soft constraint k is violated on computer i. $d_{ik} = \begin{cases} 0 & if\ soft\ constraint\ k\ on\ computer\ i\ is\ violated \\ 1 & if\ soft\ constaint\ k\ on\ computer\ i\ is\ not\ violated \end{cases}$ is an indicating function that specifies which constraint is violated. In the rest of paper, the terms of constraint and its violation only refer to soft constraint.

Objective 3: Maximizing Usability

Usability is defined as "the extent to which a product can be used by users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [19]. In our case of study, usability measures the degree to which a user is satisfied with a particular software assignment. Compared with the above two objectives, usability is relatively hard to evaluate since it can not be directly measured. While various metrics and models have been proposed to measure software usability, here we use the System Usability Scale (SUS) as an example to illustrate how to measure usability of a software assignment. SUS [20] is technology-independent scale for usability and has been tested on hardware, software, websites, etc. The standard questionnaire of SUS contains 10 items and asked the users to rate those items on a 5 points likert scale (from strongly agree to strongly disagree). A higher SUS score indicates better usability. Sample items in SUS include (The design of the questionnaire is out of scope of this paper but can be reviewed at [20]):

- I found the software unnecessarily complex.
- I thought the software was easy to use.
- I think that I would need the support of a technical person to be able to use this software.
- I found the various functions in this software were well integrated.

To quantitatively represent the overall usability of software assignment, we can collect SUS score u of every available software j from the corresponding user of computer(node) i and calculate the global usability score U by just taking the sum of all the rating scores of the software for every user/node. Thus the third objective is to maximize the overall usability score U. Assuming there are n computers and k available software, we have:

$$Maximize \qquad U = \sum_{i}^{n} \sum_{j}^{k} u_{ij} \tag{4}$$

IV. Ant Colony Optimization

A. Finding the Pareto Optimal Solutions

In general, the multi-objective version of our problem is much more difficult to solve than a single-objective one as the goals presented in a multi-objective optimization problem may be conflicting. For example, installing user-friendly software on a particular computer increases the overall usability of an assignment. However, it may violate certain security policy and decrease the survivability of the network. Finding the best solution for a multi-objective problem while satisfying all objectives simultaneously is often impossible.

Indeed, in a multi-objective setting, assignment solutions often cannot be simply compared with each other—solutions that are considered best in terms of one objective but worse in other objectives. Such solutions are called *Pareto optimal* solutions (or non-dominated solutions)[21]. A solution is Pareto-optimum when there are no other solutions that dominate it. The dominance relation can be defined as: given any two solutions S_1 and S_2, S_1 is said to dominate S_2, if both the following conditions are true (1) the solution S_1 is no worse than $S_2(f(S_1)f(S_2))$ for all objectives; (2) S_1 is strictly better than $S_2(f(S_1) \succ f(S_2))$ in at least one objective. If any of the above condition is violated, the solution S_1 does not dominate the solution S_2. If S_1 dominates S_2, then S_1 is the non-dominated solution. This concept can be extended to identify a non-dominated set of solutions from a set of solutions.

Given a non-dominated solution, no improvement is possible in any objective without sacrificing at least one of the other objectives. Hence, non-dominated solutions are considered equally good without preferences of the objectives. For our problem, instead of finding a single best solution that satisfies all the objectives simultaneously, the ultimate goal becomes to identify the Pareto optimal set that contains all non-dominated solutions to the problem. However, finding a complete Pareto optimum set of solutions is not easy. When mapping to a graph multi-coloring problem, even the single-objective software assignment problem becomes NP-hard hard[3]. Hence, in this study we present an Ant Colony Optimization (ACO) based algorithm to solve the multi-objective software assignment problem, in order to obtain the Pareto optimal solutions efficiently. Since the determination of a complete Pareto optimal set can be very difficult due to the computational complexity, here we aim to identify the approximate Pareto set.

B. Ant Colony Strategy

The ACO algorithm was first proposed by Dorigo [22], which is inspired by the foraging behavior of real ants in the wild. When ants search for food, they mark their trails with pheromone, as a way to communicate with each other. The pheromone on the path will guide other ants to the food source. Suppose there are two possible paths, the probability that an ant selects one path over the other is then based on the pheromone level of that path. Ants prefer to follow the path with stronger pheromone. Paths that are more frequently traveled become more attractive and will be used more often by the following ants. As the pheromone evaporates over time, less desirable paths become much more difficult to be detected, which further decreases their usefulness.

The foraging behavior of ants finding foods can be described in a more computational way as a multi-agent solution to a shortest-path optimization problem. In the ant algorithm, each artificial ant is considered as a computational agent and multiple ants work cooperatively in exploring the solution space for optimality.

ACO algorithms are essentially construction algorithms: every ant in the algorithm generates a complete solution starting from a null one and adding solution elements step by step. The element of a solution is a particular move of the ant, which has associated two kinds of information that guide the moving of artificial ants:

1. *Heuristic information*: a measure of the heuristic preference for a move. This information, which is known as priori knowledge to the algorithm, is specific to the problem. For example, when considering the goal of maximizing survivability in the software assignment case, heuristic information can be measured as the increased severity caused by assigning a specific software to a given node.

2. *Pheromone trail*: a measurement of the accumulative pheromone deposition for a move. This information is the posteriori knowledge of the algorithm. It is the "so far" learned preference and can be modified during the algorithm's run.

To generate Pareto solutions to the software assignment problem, the algorithm works in an iterative fashion by successively replacing the current obtained non-dominated solutions by better non-dominated solutions. In each iteration, a group of assignment solutions are generated by a colony of ants. This group of solutions, along with the non-dominated ones from previous iterations, are compared against each other by which a new set of non-dominated solutions can be identified. The process repeats until the termination criterion is satisfied.

V. Our Algorithm for Software Assignment

Generally, any ant algorithm must specify the following elements: 1) Construction of the solutions; 2) Heuristic information; 3) Pheromone updating rules; 4) Selection probability; 5) Termination condition. Specific implementations of these elements result in distinct ant algorithms. Below we describe our methods for substantializing these elements in our problem setting.

A. Algorithm Description

Construction of the Solutions: As we have mentioned, a move of the artificial ant on the graph is considered as an element of the solution. We denote a move by $m_v = (s_i, v_j)$, representing that an ant moves from somewhere to a vertex v_j and assigns v_j a software s_i. Figure 1 shows a solution construction process by consecutive moves of an ant. By moving from vertex to vertex on the graph, the assignment solution can be built incrementally. More specifically, a move actually consists two sub-actions: 1) move to a vertex, 2) assign software on that vertex. Hence two decisions have to be made at each move: the choice of the next vertex to move toward and the choice of the software to assign. In this algorithm, the selection of the next vertex to move is performed based on a pre-defined procedure, whereas the choice of the software for it is made probabilistically, depending on a combination of the heuristic information and pheromone information.

Assume that we maintain a candidate list of nodes that are available to be colored. Initially, the candidate list contains all the nodes whose degree are non-zero. We adopt the recursive largest first (RLF) principle to choose vertex at each step, in which vertices are ordered based on their current degrees, and the one with the current maximum degree is then picked for coloring. Here we re-define the concept of degree of a vertex

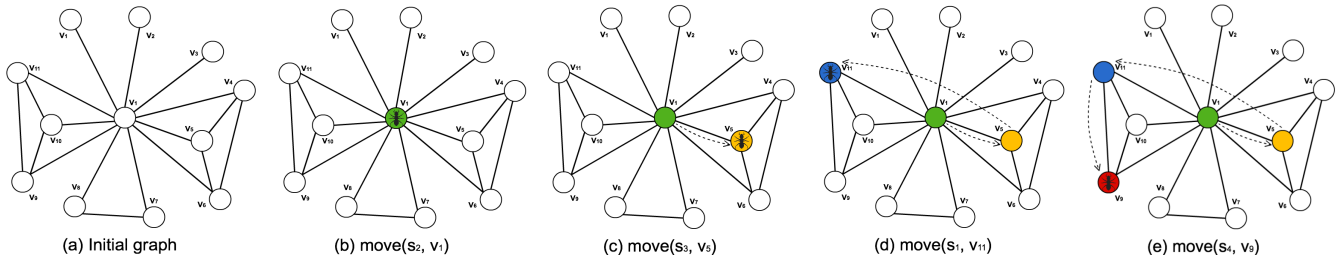

Fig. 1. Example of how an artificial ant constructs solution by moving from node to node.

v_j as $deg'(v_j) = deg(v_j) * w'_j$, where $deg(v_j)$ is the original degree of v_j and w'_j is the (remaining) weight of v_j. Initially, the weight of v_j is w_j. Every time the vertex is assigned with a software, its weight is decreased by 1 and nodes in the list needs to be re-ordered. A vertex will be removed from the list if its weight becomes 0. A complete assignment solution is obtained when the candidate list becomes empty, meaning that all the vertices now are fully assigned. Although we employ RLF to order nodes, the actual sequence of node coloring is not unique: for vertices with the same degree, the selection among them will be made randomly.

Heuristic Information: Ants refer to heuristic information while determining whether to assign software s_i to vertex v_j. The heuristic information pertaining to a move, e.g., assign s_i to v_j, is denoted by η_{ij}, which indicates the desirability of a move and can be calculated by measuring how well the software fits the given vertex with respect to a specific objective function. For each s_i on v_j, the values of attractiveness η_{ij} are computed and each corresponds to an objective function.

$$\eta_{ij}^S = 1/sv_{ij} \tag{5}$$

$$\eta_{ij}^P = 1/\sum_k P_{jk} \tag{6}$$

$$\eta_{ij}^U = u_{ij} \tag{7}$$

in which sv_{ij} represents the risk resulted by assigning s_i to v_j, and η_{ij}^S can bias the exploration of ants in favor of maximizing survivability by introducing least severity. $\sum_k P_{ik}$ represents the penalty of violating particular predefined constraint caused by assigning s_j on v_i, and η_{ij}^P bias the exploration of ants in favor of maximizing feasibility by violating relatively milder constraints, even none if possible. The heuristic related to usability is more straightforward: u_{ij} guides the assignment towards higher acceptance rate.

Pheromone Updating Rules: Pheromone represents the desirability of ants to assign software s_i to vertex v_j. In the problem considered, it represents the desirability of ants to choose each of the software options at each of the vertex. Therefore, it is represented by a pheromone matrix $PM = \{\tau_{ij}\}$, where τ_{ij} is the pheromone element corresponding to assigning s_j to vertex v_i. The value of τ_{ij} is initially set to 1.

In our case, three pheromone matrices of identical size are used, each of which corresponds to an objective function. Thus, there is a severity-related matrix PM^S, a penalty-related matrix PM^P and a usability-related matrix PM^U. To mimic pheromone evaporation and deposition, the values of elements in PM are updated separately based on each of the corresponding objectives.

$$\tau_{ij}^S = (1-\rho) * \tau_{ij}^S + \Delta\tau_{ij}^S, \Delta\tau_{ij}^S = 1/SV_k \tag{9}$$

$$\tau_{ij}^P = (1-\rho) * \tau_{ij}^P + \Delta\tau_{ij}^P, \Delta\tau_{ij}^P = 1/P_k \tag{10}$$

$$\tau_{ij}^U = (1-\rho)ast\tau_{ij}^U + \Delta\tau_{ij}^U, \Delta\tau_{ij}^U = U_k \tag{11}$$

To escape from local optima, additional procedure called *pheromone evaporation* is included. Pheromone evaporation is the process by means of which the amount of already deposited pheromone decreases over time. It favors the exploration of new search space areas and is an effective method to avoid convergence to local optimum. A uniform evaporation factor ρ is applied, which takes the value between (0, 1). At the end of each iteration of the algorithm, τ_{ij} will be updated. Clearly, if the ant did not assign software j on vertex i during the current iteration, then $\Delta\tau_{ij} = 0$. Otherwise, $\Delta\tau_{ij}$ must be calculated. Since no exact optimal solution exists with respect to all objective functions, all non-dominated solutions are considered as best solutions for our multi-objective software assignment problem. For pheromone deposition in our case, the increment $\Delta\tau_{ij}$ is calculated separately in each of the matrices. SV_k is the highest possible risk level of the solution generated by ant k, P_k is the total penalty score of the solution completed by ant k, and U_k represents the overall acceptance rate of the solution completed by k. This rule tries to increase the learning of ants.

The pheromone updates are performed by ants with a globally non-dominated solution. As such, it imposes a strong selection pressure given that ants have to find a non-dominated solution not only locally but globally in order to be allowed to update the pheromone matrices.

Selection Probability: In the implementations of ant algorithms, an ant chooses a move to go (i.e. assign software s_j to vertex v_i) based on a probabilistic decision rule that rationally combines two factors: the desirability/attractiveness of that move η_{ij} (heuristic) and the quantity of pheromone information τ_{ij} (pheromone). There are different methods in literatures to combine these factors. We have modified the method proposed in [23] to calculate the probability.

$$P_{ij} = \frac{[(\tau_{ij}^S)^{\lambda_1}(\tau_{ij}^P)^{\lambda_2}(\tau_{ij}^U)^{\lambda_3}]^\alpha[(\eta_{ij}^S)^{\lambda_1}(\eta_{ij}^P)^{\lambda_2}(\eta_{ij}^U)^{\lambda_3}]^\beta}{\sum_{j\in M}[(\tau_{ij}^S)^{\lambda_1}(\tau_{ij}^P)^{\lambda_2}(\tau_{ij}^U)^{\lambda_3}]^\alpha[(\eta_{ij}^S)^{\lambda_1}(\eta_{ij}^P)^{\lambda_2}(\eta_{ij}^U)^{\lambda_3}]^\alpha} \quad (12)$$

This probability distribution is biased by the parameters α and β, which balance the relative influence of the heuristic and the pheromone, respectively; λ_i regulates the relative importance of different objectives, $\lambda_i \in (0, 1)$.

Termination Condition: The solutions of each iteration are recorded and compared to previously obtained non-dominated solutions, so as to update the current Pareto set. The algorithm stops if the current non-dominated solutions have not been improved for the last iteration or the number of iterations meets the maximum number of iterations (an algorithm parameter).

B. Stepwise Procedure of the Algorithm

Our proposed algorithm uses multiple colonies to solve the software assignment problem. It starts by generating Max_Iter ant colonies, each consisting of $nAnts$ ants. The colonies are used one after another to move through the network. For each colony, each ant moves through nodes one after another following the RLF strategy, choosing an appropriate software on each node based on the probability decision rule represented by Eq. (11). By moving from node to node, all nodes can be completely assigned by that ant, and the assignment result represents a solution. The quality of this solution is then evaluated in terms of the three objective functions given by Eqs. (2), (3) and (4) and compared against each other in order to identify non-dominated solutions of the colony (not the entire problem). Ants that generate the non-dominated solutions of the colony are allowed to deposit pheromone increments to the pheromone matrices according to Eqs. (8), (9) and (10). In addition, before the next colony of ants start their explorations, a uniform evaporation rate is applied across elements in three pheromone matrices, The next colony then use the updated pheromone matrices to tour the network. This process repeats until the last colony has completed the coloring process and the current Pareto set generated is taken as the final solutions.

Initialize

1. Set values of parameters α, β, ρ, $nAnts$ and Max_Iter

2. Initialize the pheromone matrixes

3. for $k = 1$ to Max_Iter

4. for $i = 1$ to $nAnts$

4.1 Select a vertex to color based on RLF rule

4.2 Calculate the desirability of all possible colors for all objective functions

4.3 Calculate the probability of all possible color by combining desirability and pheromone information from PMs

4.4 Assign the most promising color

4.5 Go to Step 4.1 until current ant builds a complete solution

Evaluation

5. Calculate objective function corresponded to the constructed solutions

6. Compare the constructed solution with other solutions to determine if it is non-dominated solution

Pheromone updating

7. Compare all solutions of all $nAnts$ ants which are marked as non-dominated and identify final non-dominate solutions

8. Update the pheromone quantity in all PMs

9. If total iteration $< max\_Iter$ go to step 3 otherwise stop

C. The Case of Parallelization of ACO

Above we assume that the ants in the proposed ACO algorithm work sequentially, but the algorithm is particularly amenable to parallelization. The parallel version of the ACO algorithm can be implemented to improve efficiency and speed up convergence. Two wide categories of parallelization exist: fine-grained strategies [24] and coarse-grained strategies[25]. In the fine-grained model, each ant runs in a dedicated processor. This strategy requires much communication between processes for exchanging and synchronizing data and thus is suitable for massively parallel systems with a specific high speed topology. In the coarse-grained model, sub-colonies of ants (in the extreme case a whole colony) run on a single processor and the information exchange among those processors is done at every certain numbers of iterations. Hence, this strategy generates much less communication overhead. One can choose either one of these strategies to adjust the degree of parallelization and implement this algorithm in a distributed environment based on available computational resources and specific time requirement.

VI. Experimental Analysis

In this section, computational experiments are carried out to test the performance of the ACO-based software assignment method.

A. Experiment Setup

We have studied the experimental performance of our algorithm on three common network models, including the random network, the scale-free network and small-world network. The choice for these particular models of networks was justified by their proliferation in real-world. Each type of network graphs has 200 nodes and the average degree equals to 4, a scale that corresponds to a medium-sized organization.

In the experiments, we assumed that there were 10 different types of software packages available; each performed a distinct functionality (e.g. word processing, web browser, etc.). Within each types of software, there were also 4 different versions (e.g., Firefox, Chrome, IE and Opera in the web browser category). So, in our settings, a total of 40 different software packages were available for allocations. To simulate the penalty scores, we randomly selected 30 combinations from the 40 available software and set them as the feasibility constraints. For each constraint, a penalty score ranges from 1 to 9 was assigned. For the measurement of usability in our experiment, we used the SUS score ranging from 1 to 5 for

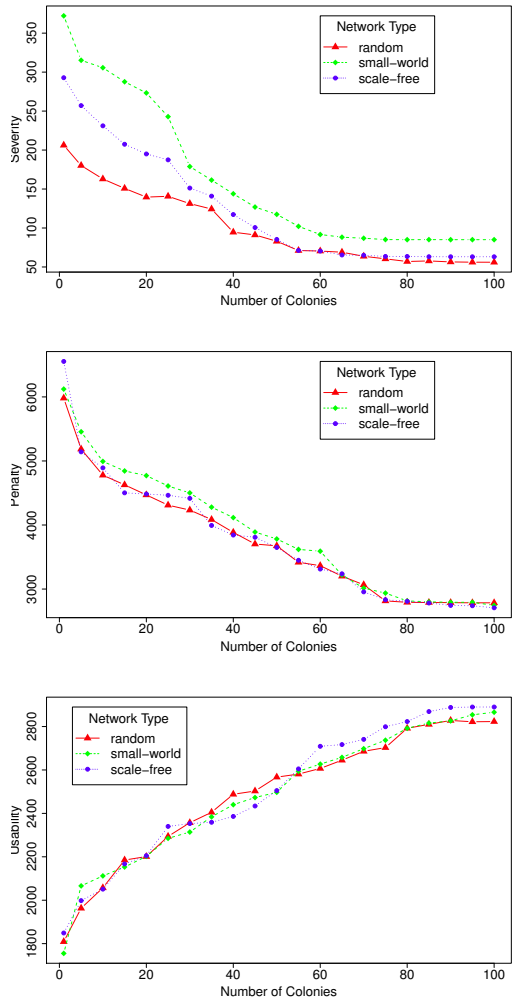

Fig. 2. Performance of the algorithm on various network topologies

each of the fictitious user. We treated the three objectives of severity, penalty and usability equally important and gave them equal relative weights with $\lambda_1 = \lambda_2 = \lambda_3 = 0.333$. We set the values of parameters $nAnts = 20$, $Max_Iter = 100$. The settings and parameters of the algorithm can be changed easily by the user according to the actual situation. Our simulations only aim to show the feasibility and effectiveness of our algorithm. We implemented the algorithm in C++ and executed on a 1.4GHz Core i5 system with 4GB DDR-2 RAM.

In the following, we performed sets of experiments to evaluate the effectiveness of the proposed algorithm: 1) on different network topologies, 2) using different combinations of parameters, 3) when solving single objective problems 4) scalability of the algorithm. We chose not to compare the proposed algorithm with the existing software assignment methods, such as [1] [2] [3], since none of the existing algorithms apply to the multiple objective setting.

B. Performance of the Algorithm on Various Graphs

The first set of experiments were conducted to test the performance of our algorithm on three types of graphs, aiming to find if the proposed algorithm works in various settings, if and how quickly the algorithm converges. For each network topology, we created a separate graph, with the y-axis indicating the score derived from the corresponding objective function and x-axis the number of iterations.

As can be seen from Figure 2, for all three graphs, after the number of iteration reaches 80, the lines become steady. This indicates that our proposed algorithm achieves stable convergence at around the 80th colony for all three types of network structures. We measured the stability of the algorithm's convergence behavior because it is one of the most important aspects of the ACO-based method. The convergence behavior of the algorithm indicates that the obtained result is a consequence of the collective intelligence of ants rather than wandering of a lucky ant.

From a further look at of Figure 2, we also notice that there is a quite big improvement of solutions at the initial stage of the process. However, as the number of iterations goes up, the improvement gradually slows down, which indicates that, as the evolution becomes mature, the results turn out to be more stable. We see that only small improvements can be achieved after the 80th iteration. It shows that the ant colony evolved to consensus objectives before the threshold maximum number of iterations is reached.

C. Effect of Parameters

In our second experiment, we test our algorithm with different parameter settings. Especially, we wanted to know i) how the ant's tendency of looking for a software option with high concentration of pheromone (α), and ii) how its tendency of choosing a software option to meet the predefined objectives (β), would affect the performance of the proposed algorithm. To determine the best values of α and β, we performed numerous simulations and noticed that the numerical relationship (greater than, less than or equal to) of α and β does affect the performance of our ACO based algorithm. So we want to determine whether α should be greater, equal or smaller than β. In this experiment, we only emphasize the test combinations in which the value set of α and β range from 1 to 2.

Figure 3 shows the results when $\alpha > \beta(\alpha = 2, \beta = 1)$, $\alpha = \beta(\alpha = 2, \beta = 2)$ and $\alpha < \beta(\alpha = 1, \beta = 2)$, respectively. From the figure, we can see that better results are achieved when $\alpha > \beta$ in our experimental settings. This suggests that pheromone plays an important role in guiding the algorithm towards convergence, although heuristic information is also important in preventing the search process from falling into local minima.

D. Performance on Solving Single Objective Problem

Since the multi-objective optimization problem is a more general form of the single objective one, our proposed method should also be effective in solving single-objective problems.

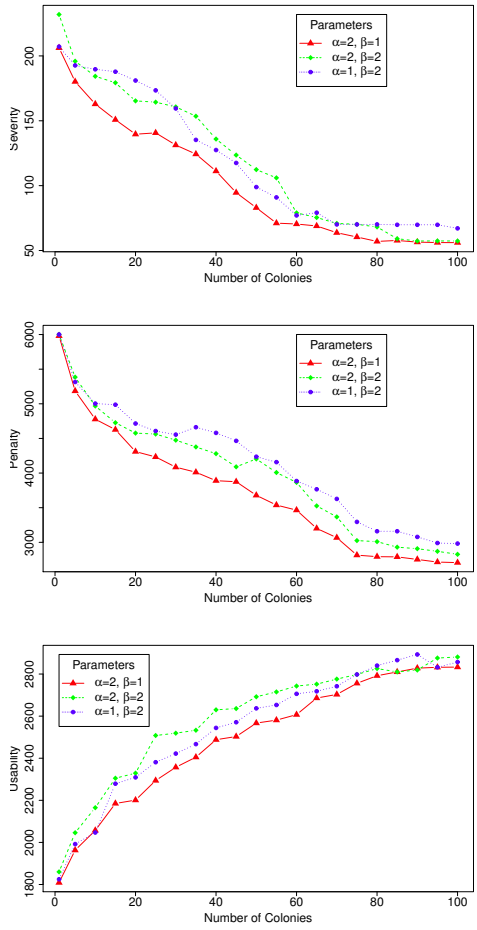

Fig. 3. Comparisons of parameters

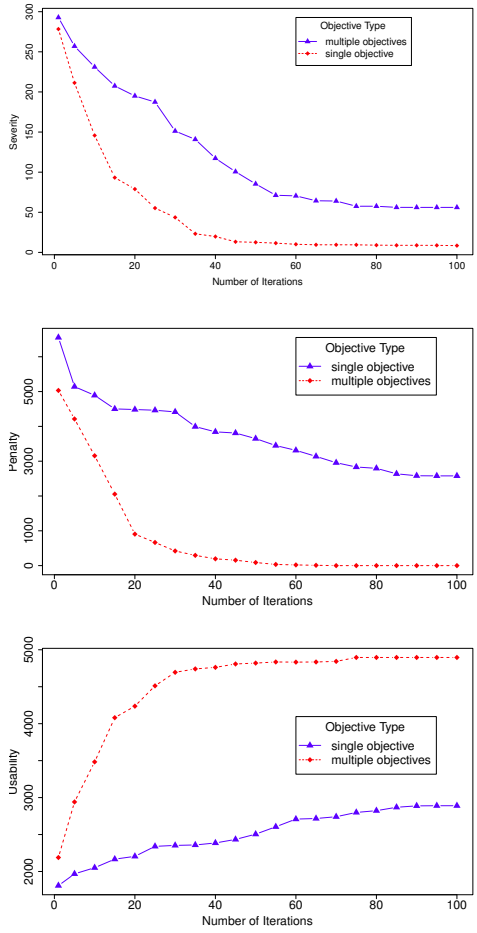

Fig. 4. Comparison of the performance for single- and multi-objective problems

In this experiment, we compared the performance of the proposed algorithm under both single and multi-objective scenarios. We specifically created three test cases in order to perform the comparisons, with each has a single objective, i.e. survivability, feasibility and usability, respectively.

Figure 4 shows the comparison results. We observe that our algorithm reaches convergence more rapidly in the single-objective cases than in the multi-objective cases. The generated assignment solutions for single objective scenarios also perform better than that for the multiple objective situations with regard to each objective function. The results are actually reasonable for both cases, because the multi-objective problem requires much more effort in finding the approximate Pareto solutions, especially when conflicting objective functions tend to restrain each other.

E. Pareto Solutions of the Problem

For better illustration, we depict the Pareto solutions generated by our algorithm in a 3-dimensions space, as shown in Figure 5. The three dimensional plot can be served as a general portfolio containing all alternative solutions. In addition, It demonstrates the trade-offs among survivability, feasibility and usability, which allows the users to choose from, depending on different priority settings. On a more extreme level, to achieve our pre-defined objectives, we next pick three solutions from all Pareto solutions. Each selected solution contains the minimum severity, minimum penalty, and maximum usability, respectively in the gained Pareto set of solutions, as shown in Table I.

TABLE I
MAXIMUM/MINIMUM SOLUTIONS ALONG EACH OBJECTIVE

Severity	*Penalty*	*Usability*
Minimum severity		
56.083	5866	2782
	Minimum penalty	
129.2961	2578	1809
		Maximum usability
70.483	5063	2973

F. Scalability and Computational Overhead

Finally, we repeated our experiments on graphs of larger sizes and measured the computational overhead introduced by this algorithm. Figure 6 illustrates the scalability performance

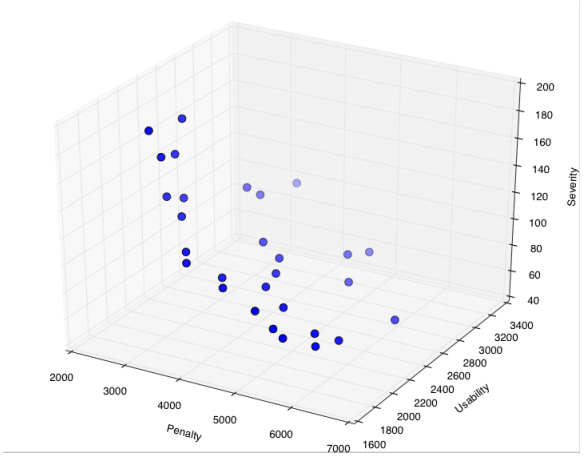

Fig. 5. Pareto optimal solutions

of the algorithm on small-world graph. As can be seen from it, the computational time required for this algorithm grows exponentially with the size of the network, but it is still feasible for medium-size networks.

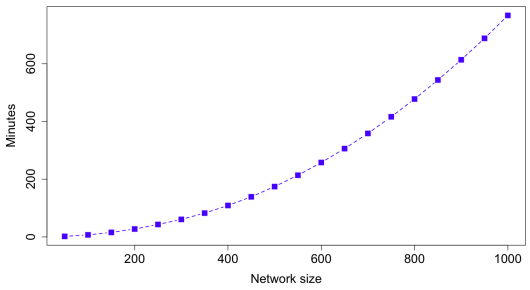

Fig. 6. Scalability of the algorithm

VII. Conclusions

In this work, we introduced a multi-objective optimization formulation for the software assignment problem. Three practical objectives are taken into account: 1) the maximization of network survivability by minimizing of the attack severity; 2) the maximization of the system feasibility by minimizing the violation of practical constraints 3) the maximization of the system usability. To find the approximate Pareto-optimal set of solutions in an efficient manner, we proposed an ACO-based algorithm for solving the software assignment problem with the goal of reducing the potential risk of a successfully attack while enhancing system feasibility and usability. To verify the performance of the proposed algorithm, computational experiments are conducted. The results show that this algorithm solves the multi-objective software assignment problem by providing a set of non-dominated solutions meeting different needs and preferences in practice. In the future, we intend to investigate a comprehensive evaluation framework for assessing and comparing different software assignment algorithms.

Acknowledgements: This work is supported in part by the NSF grant CCF-1320605 and Science and Technology Planning Project of Guangdong Province, China (2014A040401027, 2012A080102007).

References

[1] A. J. O'Donnell and H. Sethu, "On achieving software diversity for improved network security using distributed coloring algorithms," in *Proceedings of ACM CCS*. ACM, 2004, pp. 121–131.

[2] Y. Yang, S. Zhu, and G. Cao, "Improving sensor network immunity under worm attacks: a software diversity approach," in *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*. ACM, 2008, pp. 149–158.

[3] C. Huang, S. Zhu, and R. Erbacher, "Toward software diversity in heterogeneous networked systems," in *Data and Applications Security and Privacy XXVIII*. Springer, 2014, pp. 114–129.

[4] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE, 1997, pp. 67–72.

[5] S. A. Hofmeyr and S. Forrest, "Architecture for an artificial immune system," *Evolutionary computation*, vol. 8, no. 4, pp. 443–473, 2000.

[6] J. Caballero, T. Kampouris, D. Song, and J. Wang, "Would diversity really increase the robustness of the routing infrastructure against software defects?" *Department of Electrical and Computing Engineering*, p. 40, 2008.

[7] E. Totel, F. Majorczyk, and L. Mé, "Cots diversity based intrusion detection and application to web servers," in *Recent Advances in Intrusion Detection*. Springer, 2006, pp. 43–62.

[8] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and A. Romanovsky, *Using diversity in cloud-based deployment environment to avoid intrusions*. Springer, 2011.

[9] F. B. Cohen, "Operating system protection through program evolution," *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[10] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 281–289.

[11] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 272–280.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits." in *USENIX Security*, vol. 3, 2003, pp. 105–120.

[13] ——, "Efficient techniques for comprehensive protection from memory error exploits." in *Usenix Security*, 2005.

[14] A. Colorni, M. Dorigo, V. Maniezzo *et al.*, "Distributed optimization by ant colonies," in *Proceedings of the first European conference on artificial life*, vol. 142. Paris, France, 1991, pp. 134–142.

[15] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 26, no. 1, pp. 29–41, 1996.

[16] C. Rajendran and H. Ziegler, "Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs," *European Journal of Operational Research*, 2004.

[17] E. Salari and K. Eshghi, "An aco algorithm for graph coloring problem," in *Computational Intelligence Methods and Applications, 2005 ICSC Congress on*. IEEE, 2005, pp. 5–pp.

[18] T. Stützle and H. H. Hoos, "Max–min ant system," *Future generation computer systems*, vol. 16, no. 8, pp. 889–914, 2000.

[19] N. Bevan, "Usability is quality of use," *Advances in human factors ergonomics*, vol. 20, pp. 349–349, 1995.

[20] J. Brooke, "Sus-a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[21] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *evolutionary computation, IEEE transactions on*, vol. 3, no. 4, pp. 257–271, 1999.

[22] M. Dorigo and M. Birattari, "Ant colony optimization," in *Encyclopedia of Machine Learning*. Springer, 2010, pp. 36–39.

[23] C. García-Martínez, O. Cordón, and F. Herrera, "An empirical analysis of multiple objective ant colony optimization algorithms for the bi-criteria tsp," in *Ant Colony Optimization and Swarm Intelligence*, 2004.

[24] I. Ellabib, P. Calamai, and O. Basir, "Exchange strategies for multiple ant colony system," *Information Sciences*, 2007.

[25] J. A. Mocholí, J. Jaen, and J. H. Canos, "A grid ant colony algorithm for the orienteering problem," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 1. IEEE, 2005, pp. 942–949.