Reliable Mission Deployment in Vulnerable Distributed Systems

Massimiliano Albanese*, Sushil Jajodia*, Ravi Jhawar†, and Vincenzo Piuri†
*George Mason University, Fairfax, VA, USA
Email: {malbanes,jajodia}@gmu.edu
†Università degli Studi di Milano, Crema, Italy
Email: {ravi.jhawar, vincenzo.piuri}@unimi.it

***Abstract*—Recent years have seen a growing interest in mission-centric operation of large-scale distributed systems. However, due to their complexity, these systems are prone to failures and vulnerable to a wide range of cyber-attacks. Current solutions focus either on the infrastructure itself or on mission analysis, but fail to consider information about the complex interdependencies existing between system components and mission tasks. In this paper, we take a different approach, and present a solution for deploying mission tasks in a distributed computing environment in a way that minimizes a mission's exposure to vulnerabilities by taking into account available information about vulnerabilities and dependencies. We model the mission deployment problem as a task allocation problem, subject to various dependability constraints. The proposed solution is based on the A^* algorithm for searching the solution space, but we also introduce a heuristic to significantly improve the search performance. We validate our approach, and show that our algorithm scales linearly with the size of both missions and networks.**

***Index Terms*—Data-driven reliability, mission-centric computing, vulnerability analysis.**

I. Introduction

Large-scale distributed systems are increasingly being adopted for a wide range of applications. The availability of dynamic and scalable pools of computational resources makes such systems particularly attractive for mission-critical applications. Despite the significant benefits, these systems – due to their complexity – are prone to a number of failures and are vulnerable to a wide range of cyber-attacks, which may have a significant impact on the success of a mission. Therefore, it is critically important to assess the dependability of computing systems and deploy computational tasks taking dependability constraints into account.

Traditional approaches to improve security include (i) designing networks with services such as intrusion detection systems, firewalls, and other network hardening tools; and (ii) developing mission tasks using security measures such as data obfuscation and memory management. However, complex interdependencies between network infrastructure, mission tasks, and residual vulnerabilities in the system are typically not taken into account. This implies that existing solutions fail to complement network and mission's security, and, as a consequence, they provide opportunities for attackers to penetrate the network and compromise a mission.

The work presented in this paper is supported in part by the Office of Naval Research under award number N00014-12-1-0461.

In contrast with traditional solutions, we propose a novel approach that enables deployment of mission tasks in distributed systems such that their exposure to network vulnerabilities is minimized. In other words, instead of further hardening the network – which may have a significant cost, with diminishing returns – we consider the current state of the system in terms of vulnerability distribution and develop a solution to deploy mission tasks in the most secure manner possible. We model our reliable mission deployment problem as a task allocation problem with a mission-centric, security-oriented objective, and develop a state-space search algorithm to solve it. We consider an incremental deployment approach since requests for mission deployment may arrive at any time. That is, when a request is received, the allocation of the new mission is decided based on the current availability of resources.

The contribution of this paper is twofold. First, we formulate the mission deployment problem as a task allocation problem subject to security and dependability constraints. Second, we propose a state-space search technique based on the A^* algorithm to find near-optimal allocations in a time-efficient manner. Finally, we introduce a heuristic to improve the performance of the search algorithm without significantly degrading the quality of the solutions.

Although the work presented in this paper represents only the first step towards a mission-centric and security-aware computational framework for distributed systems, the results we present are encouraging and motivate further research in this direction. Additionally, from a practical perspective, the proposed approach can be readily applied in all those real-world scenarios where it is not possible to remediate all existing vulnerabilities, and missions have to be executed on complex systems with multiple and interdependent residual vulnerabilities. In such scenarios, one may want to identity an execution plan that minimizes the risk that residual vulnerabilities may impact the mission and eventually compromise its successful completion. In order to be effective, the proposed approach must be integrated with existing techniques for identifying network vulnerabilities and generating attack models – such as the attack graphs generated by Cauldron [1] – as well as techniques for assessing the risk that residual vulnerabilities may pose to each element of the computing infrastructure [2].

The paper is organized as follows. Section II discusses related work, and Section III presents some preliminary

definitions. Section IV provides a formal statement of the mission deployment problem, whereas Section V presents our solution. Finally, Section VI reports our experimental results, and Section VII gives some concluding remarks.

II. Related Work

The problem of reliable mission deployment in vulnerable networks has not been sufficiently studied in the literature. The approach presented in this paper relies on assessing the effect that interdependent vulnerabilities may have on various, possibly interdependent network assets. Jakobson [3] presents a method for analyzing the impact of attacks on a given mission using extended dependency graphs. Albanese *et al.* [4] use attack graphs to define a network hardening scheme that considers the impact of interdependent hardening actions. Mehta *et al.* [2] propose a ranking scheme for assigning probabilities to the states of an attack graph, thus providing a measure of their vulnerability.

A line of research relevant to our work consists in integrating the security requirements of applications within resource allocation and scheduling processes. Jhawar *et al.* [5] model the security requirements of an application as additional constraints and present a greedy heuristic to perform VM allocation in cloud computing and minimize energy consumption. Similarly, Xie *et al.* [6] and Xiong *et al.* [7] define heuristics that balance security and performance.

In general, task allocation in distributed systems has been widely studied, particularly with the objective of minimizing the make-span. Allocation strategies to improve reliability and load balancing have also been considered [8]. Given the NP-hardness of the task allocation problem, most solutions are based on heuristics, meta-heuristics, or mathematical programming. The min-min, max-min, and duplex heuristics are leveraged in [9], [10], whereas [11] presents a graph-theoretic heuristic for task allocation. Theoretical optimality guarantees of these algorithms are generally poor. However, some heuristics offer practical suboptimal solutions. Meta-heuristics such as genetic algorithms and simulated annealing have also been widely used [7]. They guarantee near optimal solutions, but the time to converge is significantly high. Another approach to task allocation consists in modeling the problem as an integer programming problem [12].

In this paper, we propose an approach to task allocation consisting in a state-space search technique based on the A^* algorithm [13]. A^* is well known for its performance and accuracy in path finding and game theory applications, but only in a few cases it has been used for task allocation [14], [15]. Our work differs significantly from previous work in that we (i) reduce the reliable mission deployment problem to a task allocation problem whose primary objective is to minimize each task's exposure to vulnerabilities in a large-scale system, (ii) provide near optimal results in a time-efficient manner, and (iii) take into account additional dependability constraints.

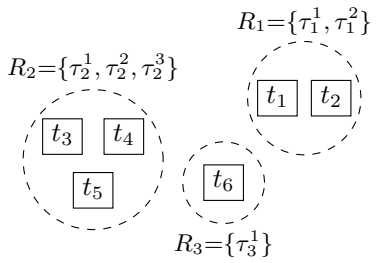

Fig. 1. Mission comprising three tasks

III. Missions in a Distributed System

In this section, we present some preliminary concepts and our assumptions on network infrastructure and missions.

A large-scale distributed system can be viewed as a large pool of interconnected physical hosts that is partitioned into multiple subnets. Hosts in each subnet are connected *locally*, with similar network configurations, and subnets are connected using routers. Despite careful security engineering, a number of exploitable vulnerabilities remain in the network, providing attackers with opportunities to penetrate the system and compromise missions. In order to ensure reliable mission deployment, we need to analyze residual vulnerabilities and attack paths, assign quantitative vulnerability scores to each host, and use this information to securely deploy mission tasks. Note that existing solutions, such as those based on attack graphs [2], can be used to assess the overall vulnerability of network assets. Therefore, we can assume that a vulnerability score V_h has been precomputed for each host h, and refer the reader to the literature for details on how this can be achieved.

Let $\mathcal{H}$={h_j}={$h_1, \ldots, h_n$} be the set of all physical hosts in the network. Since we want to perform incremental deployment of missions, we only consider currently unused resources for each host. A physical host $h \in \mathcal{H}$ is characterized by a vector $\overrightarrow{h}$=($h[1], h[2], \ldots, h[d], h[d+1]$), where the first d dimensions represent the residual *capacity* of each resource (e.g., CPU, memory), and the last dimension is the host's *vulnerability score* V_h. We normalize the elements of $\overrightarrow{h}$ to values between 0 and 1. For instance, $\overrightarrow{h}$=(CPU,mem,V)=$(1, 1, 1)$ implies that CPU and memory are fully available and the host is extremely vulnerable.

In order to keep our approach independent of specific formalisms for representing missions, we simply define a mission M as a set of tasks M={$\tau_1, \ldots, \tau_m$}. A mission is *successful* if all its tasks are correctly executed, possibly within a specified amount of time. Intuitively, a mission is a composition of interacting tasks with some tasks being more critical than others. Therefore, we assume the existence of a *criticality function* $c : M \rightarrow \mathbb{R}$ that assigns a criticality value $c(\tau)$ to each task $\tau \in M$. The actual definition of the criticality function c depends on the specific formalism used to model missions. In general, to improve the *fault tolerance* and *resilience* of a mission, one may replicate the most critical tasks using a replication strategy.

Definition 1 (Replication strategy): Given a mission $M = \{\tau_1, \ldots, \tau_m\}$ and a criticality function c, a replication strategy associates each task $\tau_k \in M$ with a set of task replicas $R_k = \{\tau_k^l\} = \{\tau_k^1, \ldots, \tau_k^{|r_k|}\}$, called the *replicated task set*, such that the following monotonicity axiom holds:

$$(\forall \tau_{k_1}, \tau_{k_2} \in M), \quad c(\tau_{k_1}) < c(\tau_{k_2}) \Rightarrow |R_{k_1}| \leq |R_{k_2}| \quad (1)$$

Intuitively, a replication strategy guarantees that more critical tasks will be assigned a higher number of replicas. In general, a replication strategy provides the semblance of a replicated task set as a single task and ensures that the task continues to execute correctly even in the presence of a given number of failures. In our framework, we define the set T of tasks to be deployed as $T = \{t_i\} = \bigcup_{\tau_k \in M} R_k$. In other words, the deployment scheme needs to allocate $|T| = \sum_{\tau_k \in M} |R_k|$ tasks (note that $|T| \geq |M|$). Figure 1 shows an example of mission with three tasks $M = \{\tau_1, \tau_2, \tau_3\}$. The replicated task sets – represented with dotted circles – are $R_1 = \{\tau_1^1, \tau_1^2\}$, $R_2 = \{\tau_2^1, \tau_2^2, \tau_2^3\}$ and $R_3 = \{\tau_3^1\}$. The set of tasks to be deployed is $T = \{t_1, \ldots, t_6\}$.

In general, one may implement specific mechanisms to make tasks more robust to vulnerability exposure and failures. Based on these mechanisms and on their intrinsic criticality, different tasks may have different levels of risk propensity. We can then assume the existence of a function $tol : M \to \mathbb{R}$ that assigns a risk tolerance value $tol(t)$ to each task t. Intuitively, higher values of $tol(t)$ imply that t can be deployed on hosts that are more likely to be compromised.

Similarly to hosts, a task $t \in T$ is characterized by a vector $\overrightarrow{t} = (t[1], t[2], \ldots, t[d], t[d+1])$, where the first d dimensions represent the task's *requirements* for specific computing resources and the last dimension is the task's risk tolerance value $tol(t)$. We normalize the elements of $\overrightarrow{t}$ to values between 0 and 1.

IV. Problem Formalization

We assume that tasks are to be executed in a virtual environment and, for each task, we can instantiate a virtual machine (VM) capable of executing that task. Given this assumption, it is clear that two subproblems need to be solved, namely: (i) mapping tasks to available VM images, and (ii) allocating VMs on available physical hosts. In order to focus on the second problem, we assume that for each task $t \in T$ there exists a virtual machine image I that can be characterized as $\overrightarrow{t}$. With this assumption, the problem reduces to allocating tasks in T to physical hosts.

Definition 2 (Task allocation): A *task allocation* is a function $a : T \to \mathcal{H}$ which maps each task $t \in T$ to a physical host $h \in \mathcal{H}$. The binary variable a_{ij}, for each $t_i \in T$ and $h_j \in \mathcal{H}$, denotes the truth value of $a(t_i) = h_j$, that is, $a_{ij} = 1$ if $a(t_i) = h_j$, 0 otherwise.

Each allocation must satisfy the following constraints to ensure the dependability of a mission: (i) consistent allocation, (ii) distribution, and (iii) vulnerability tolerance. We discuss these constraint in detail in the following.

Consistent allocation. The following properties must hold.

TABLE I
Example scenario for mission deployment

Network		Mission	
$h_j \in \mathcal{H}$	$\overrightarrow{h}$ (cpu, V)	$t_i \in T$	$\overrightarrow{t}$ (cpu, tol)
h_1	0.5, 0.3	t_1	0.4, 0.2
h_2	0.7, 0.1	t_2	0.4, 0.2
h_3	0.3, 0.2	t_3	0.3, 0.4
h_4	0.5, 0.2		

TABLE II
Increases in vulnerability scores

$\Delta V_{t_i,h_j}$	h_1	h_2	h_3	h_4
t_1	0.3	0.1	0.1	0.2
t_2	0.1	0.2	0.1	0
t_3	0	0.2	0.1	0.1

$$(\forall t_i \in T) \quad \sum_{h_j \in \mathcal{H}} a_{ij} = 1 \quad (2)$$

$$(\forall h_j \in \mathcal{H}, 1 \leq x \leq d) \quad \sum_{t_i \in T} a_{ij} \cdot t[x] \leq h[x] \quad (3)$$

Equation 2 implies that each task must be allocated on a single physical host (however, each host can accommodate multiple tasks). Equation 3 implies that the amount of resources consumed on a single host cannot exceed the total residual capacity of that host in any dimension[1].

Distribution. All replicas of a task must be allocated on different hosts to avoid single points of failure.

$$(\forall \tau_k \in M)(\forall \tau_k', \tau_k'' \in R_k) \quad a(\tau_k') \neq a(\tau_k'') \quad (4)$$

Vulnerability tolerance. A task t can be mapped only to hosts h whose vulnerability score $V_h = h_j[d+1]$ is less than the vulnerability tolerance $tol(t) = t_i[d+1]$ of that task.

$$(\forall h_j \in \mathcal{H}, t_i \in T) \quad t_i[d+1] \geq a_{ij} \cdot h_j[d+1] \quad (5)$$

Our objective is to minimize a mission's exposure to vulnerabilities. Note that each time a task t_i is allocated on a host h_j, new vulnerabilities are potentially introduced on the host. As a consequence, the vulnerability score of h_j may increase by an amount $\Delta V_{t_i,h_j}$[2]. Note that, although multiple hosts may have similar configurations and, consequently, similar vulnerability scores, their vulnerability scores may vary significantly at runtime, as tasks are dynamically allocated and deallocated. Let $V_{h_j}^*$ denote the vulnerability score of host h_j after mission deployment. Our objective is to find, among all possible allocations $a \in \mathcal{A}$, the allocation that minimizes the largest $V_{h_j}^*$ amongst all the hosts involved in the mission, that is

$$\min_{a \in \mathcal{A}} \max_{h_j \in \mathcal{H} | \exists t_i \in T, a(t_i) = h_j} V_{h_j}^* \quad (6)$$

Example 1: Consider a mission with two tasks $M = \{\tau_1, \tau_2\}$, where $\mathcal{R}_1 = \{\tau_1^1, \tau_1^2\}$ and $\mathcal{R}_2 = \{\tau_2^1\}$, and a

[1]For the sake of presentation, we are omitting to consider the resources consumed for managing the virtualized environment.

[2]The amount $\Delta V_{t_i,h_j}$ can be estimated with the same approach used to estimate the baseline vulnerability score V_h.

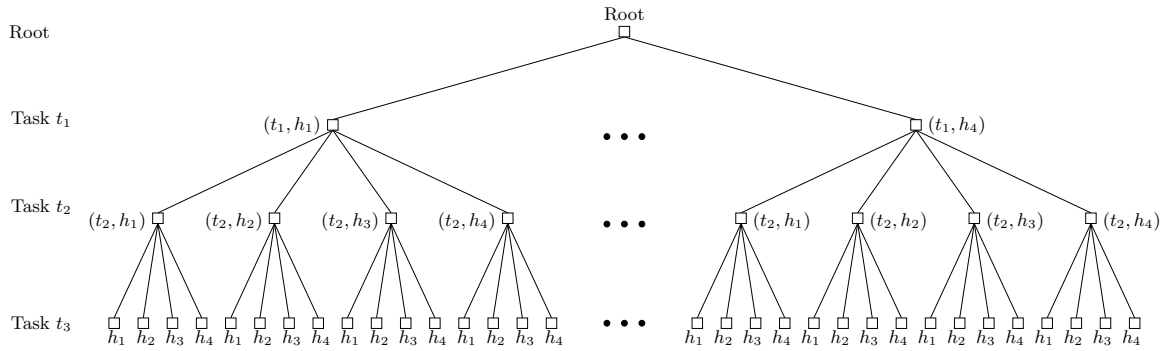

Fig. 2. State-space tree for a network with four hosts and mission with three tasks

network with four hosts $\mathcal{H}=\{h_1, \ldots, h_4\}$. This implies a distribute constraint between tasks t_1 and t_2. For simplicity, we assume that hosts have a single resource (e.g., CPU). Table I provides details about available CPU capacity and vulnerability score of each host, as well as CPU requirements and risk tolerance of each task. Instead, Table II shows the values of $\Delta V_{t_i,h_j}$ for all tasks and hosts. We aim at allocating tasks such that all constraints are satisfied and the overall risk for the mission is minimized. Consider the vulnerability distribution of Table I. Since host h_2 is the least vulnerable host, one might think that mapping all the tasks on this host can provide the best solution. Assuming that t_1 is mapped to h_2, the only feasible allocation for task t_2 is on host h_4 since h_1 does not satisfy the vulnerability constraint and h_3 does not have enough capacity. Hosts h_1, h_2, and h_3 satisfy all the constraints for task t_3 and any of them can be chosen. However, in a realistic scenario, the order of task allocation is important (e.g., if we allocate task t_2 on host h_2 first, there is no feasible mapping for task t_1).

V. Mission Deployment

In this section, we formulate our task allocation problem as a state-space search problem, and solve it by adapting the well-known A^* algorithm [13]. To the best of our knowledge, an A^*-based algorithm has not been used before to *support the dependability constraints* defined in Section IV and *improve the security of task deployment*, as we do in this paper. We first present the data structure supporting the exploration of the solution space and the basic exploration operation, and, finally, our heuristic approach to A^* exploration.

A. States and Generation of Child States

To enable A^* exploration, the overall state-space is represented as a tree where each node, called *state*, represents a possible choice for allocating a single task, in addition to the tentative allocations performed on previous levels in the tree. Figure 2 illustrates the complete state-space tree for the scenario described in Example 1. We now briefly describe the components of this representation.

State. A state s is a possible choice for allocating a task t_i on a host h_j. A state is represented by a pair $(t_i, h_j) \in T \times \mathcal{H}$.

Algorithm 1 $getSuccessors(s)$

Input: Current state s representing partial allocation (t, h)
Output: Set of successors S

1: // T^* is the list of tasks sorted in the increasing order of tol value; $|T^*| = |T|$. $\mathcal{H}$ is the set of hosts in the network. T^* and $\mathcal{H}$ are globally available
2: $S \leftarrow \emptyset$
3: **if** $s = root\ state$ **then**
4: $t^* \leftarrow T_1^*$ // Choose the first task from T^*
5: **else**
6: $t^* \leftarrow T_{t+1}^*$ // Choose the next task from T^*
7: **end if**
8: **for all** $h_j \in \mathcal{H}$ **do**
9: **if** $t^*[d+1] \geq h_j[d+1]$ **then**
10: // Vulnerability tolerance constraint is satisfied
11: **if** $t^*[x] \leq h_j[x]\ \forall x \in [1, d]$ **then**
12: // Capacity constraint is satisfied
13: **if** $(t', h_j) \notin$ CLOSE s.t. $(t', t^*) \in R_k$ **then**
14: // Distribute constraint is satisfied
15: $S \leftarrow S \cup \{(t^*, h_j)\}$
16: **end if**
17: **end if**
18: **end if**
19: **end for**
20: **return** S

Root state. The root state is the initial state from which the algorithm starts, with no task being allocated yet.

Operation. Given a state s, an operation of the A^* algorithm generates the set of feasible child states for s. This is done by choosing the next task to be allocated and considering all compatible hosts, as described in the following.

Solution path. A solution path is a path from the root state to the first leaf state reached during state-space exploration. A leaf state corresponds to a complete allocation.

Goal state. A goal state is a state in which all the tasks have been allocated.

The set T of tasks to be allocated is initially sorted in increasing order of risk tolerance tol. The i-th task in the sorted list corresponds to the i-th level of the state-space tree. The number of levels in the tree is equal to the number of tasks to be allocated plus the root state (level 0). We now describe our approach to generating the child states of a given state s, which is done using the *getSuccessors* function outlined in Algorithm 1. This function takes the current state $s = (t, h)$ as input and performs the following operations.

a) The task t^* following t in the ordered list is selected. In the case of the root state, the first task from the list is

selected (lines 3−7).

b) The set of hosts $h_j \in \mathcal{H}$ that satisfy all mapping constraints (see Section IV) with respect to task t^* are shortlisted. Specifically, we first enforce the *vulnerability tolerance constraint* in order to select the hosts $h_j \in \mathcal{H}$ that satisfy $t^*[d+1] \geq h_j[d+1]$. Next, we enforce the *capacity constraint* in order to select the hosts that satisfy the resource requirements for task t^* w.r.t. all d dimensions, that is, the hosts h_j such that $(\forall x \in [1,d])\, t^*[x] \leq h_j[x]$. Finally, we enforce the *distribute constraint* w.r.t. all the hosts by verifying that there does not exist a state (t', h_j) in the current solution path such that t^* and t' belong to the same replicated task set R_k. As we show later, this condition can be verified by looking at the "visited" states in the `CLOSE` data structure (lines 8−14).

c) For each shortlisted host h_j, a state (t^*, h_j) is generated and added to the child states of s (line 15).

To avoid an exponential explosion of the search space, when a sub-optimal solution is acceptable, only the p least vulnerable hosts $h \in \mathcal{H}$ are considered, where p is a user-defined parameter. This makes our A^* based exploration a heuristic approach. By considering tasks in the increasing order of their tol values, we ensure that the tasks with the most stringent security requirements are allocated first. This avoids the problem of reserving highly secure resources for less critical tasks. Moreover, this choice seems reasonable since other parameters, such as resource capacity, are typically abundantly available in large-scale networks.

B. State-Space Search Scheme

The previous subsection provided a general overview of the application of the A^* algorithm to task allocation. Our search scheme builds on the A^* algorithm to find a solution path that minimizes the mission's exposure to vulnerabilities.

In general, the A^* algorithm uses a cost function to determine the order in which the states must be considered for expansion [13], [14]. This cost function has the form $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of the path from the root state to the current state s, and $h(s)$ is a lower-bound estimate of the cost of the path from s to a goal state. The algorithm reduces the number of state expansions when compared to a simple greedy algorithm, and can find optimal solutions when an *admissible* heuristic is used to estimate $h(s)$. We define our cost function as follows

$$f_{vul}(s) = g_{vul}(s) + h_{vul}(s) \tag{7}$$

where $g_{vul}(s)$ denotes the aggregate vulnerability score associated with the allocation path from the root state to the current state s, and $h_{vul}(s)$ estimates the minimum additional vulnerability associated with completing the allocation of the mission's tasks. We can compute $g_{vul}(s)$ as follows

$$g_{vul}(s) = g_{vul}(parent(s)) + V_{h_j} + \Delta V_{t_i,h_j} \tag{8}$$

Algorithm 2 $estimateCost(s^*)$

Input: Present state s^* representing partial allocation (t, h)
Output: $h_{vul}(s^*)$, lower bound estimate of cost from s^*

```
1:  g_value ← ∅ // Heap data structure to temporaily store g_vul values
2:  state ← s*
3:  h_vul ← 0, cost ← 0
4:  while state ≠ goal state do
5:      S ← getSuccessors(state)
6:      for all s** ∈ S do
7:          cost ← (V_h + ΔV_{t,h})
8:          push_heap(s**, cost, g_value)
9:      end for
10:     s' ← pop_heap(g_value) // Pop the state with minimum cost value
11:     h_vul ← h_vul + cost
12:     state ← s'
13:     g_value ← ∅
14: end while
15: return h_vul
```

where $g_{vul}(parent(s))$ denotes the aggregate vulnerability score associated with the allocation path leading to the parent state of s and $V_{h_j} + \Delta V_{t_i,h_j}$ denotes the updated vulnerability score of host h_j after the allocation of task t_i. If s is the root state, then we initialize $g_{vul}(s)$ to 0.

In the simplest case, the lower bound estimate can be defined as $h_{vul}(s) = 0$. In this case, we assume a *uniform cost search* and, as a consequence, the algorithm may expand and visit a higher number states before reaching an optimal goal state. In order to improve the performance of the search process, heuristics are typically adopted. Algorithm 2 outlines our heuristic approach to estimate $h_{vul}(s^*)$, when the traversal algorithm is at state s^* of the state-space tree.

a) In the current state s^*, we use an *operation* to obtain the set S of feasible child states (line 5).

b) Since our cost function is the aggregate vulnerability score associated with complete task allocation, we compute $V_{h_j} + \Delta V_{t_i,h_j}$ for each state in S (lines 6−9).

c) To ensure that $h_{vul}(s)$ is a lower bound, we select the state with the smallest value of $V_{h_j} + \Delta V_{t_i,h_j}$ and temporarily mark it as the current state (lines 10−13).

d) We repeat steps a, b, and c until a *goal state* is reached, and return h_{vul} as the aggregate vulnerability score along the path leading to this goal state (line 15).

Note that h_{vul}, computed using an admissible heuristic, improves the performance of the state-space search without influencing the final results of the search. We demonstrate the effectiveness of our heuristic through Example 2 as well as through the experimental evaluation of Section VI.

We now describe our state-space tree *traversal scheme* that generates the solution path that minimizes a mission's exposure to vulnerabilities. Two data structures `OPEN` and `CLOSE` (implemented as heaps) are used for making traversal decisions: `OPEN` contains the set of states that are generated using the *operation* but not yet visited, and `CLOSE` contains the states that have been visited already. Each entry in `OPEN` and `CLOSE` consists of a state s and its $f_{vul}(s)$ value. We start from the root state, where no task is allocated, push it in `OPEN`, and perform the following steps until a complete solution is obtained or `OPEN` is empty.

Algorithm 3 *StateSpaceSearch*

```
1:  // OPEN, CLOSE are heap data structures that are available globally
2:  // f_vul(root state)=0, g_vul(root state)=0
3:  push_heap(root state, 0, OPEN)
4:  while OPEN ≠ ∅ do
5:      s ← pop_heap(OPEN)
6:      // Pop the state with minimum f_vul(s) value
7:      if s = goal state then
8:          // Search found complete solution
9:          push_heap(s, f_vul(s), CLOSE)
10:         constructSolution(s)
11:         return True
12:     else
13:         S ← getSuccessors(s)
14:         for all s* ∈ S do
15:             new_g_vul ← g_vul(s) + (V_h + ΔV_{t,h})
16:             // s* represents a partial allocation (t, h)
17:             if (s* ∈ OPEN) ∨ (s* ∈ CLOSE) then
18:                 if g_vul(s*) ≤ new_g_vul then
19:                     continue
20:                 end if
21:             end if
22:             g_vul(s*) ← new_g_vul
23:             f_vul(s*) ← g_vul(s*) + estimateCost(s*)
24:             push_heap(s*, f_vul(s*), OPEN)
25:         end for
26:         push_heap(s, f_vul(s), CLOSE)
27:     end if
28: end while
29: return False
```

1. Pop the state s with minimum $f_{vul}(s)$ from OPEN (line 5).
2. If s is a goal state, construct the final solution by traversing the tree backwards from the goal state to the root (lines 7−11).
3. If s is not a goal state, generate its successors (child states) using an *operation*.
4. For each successor s^* of s
   a. Calculate the updated cost (aggregated vulnerability score) new_g_{vul} for s^* (line 15).
   b. If an entry corresponding s^* already exists in either OPEN or CLOSE, and its real cost is less than that of the current successor, drop the current successor (and consider the next one) since the same state has already been reached with lower cost. Otherwise, continue with the next step (lines 17−21).
   c. Estimate the lower bound $h_{vul}(s^*)$ using the *estimateCost* function and compute $f_{vul}(s^*) = g_{vul}(s^*) + h_{vul}(s^*)$ (line 23). Note that the h_{vul} value for a leaf node is zero.
   d. Push the pair $(s^*, f_{vul}(s^*))$ into OPEN (line 24).
5. Push s into CLOSE since it has been visited (line 26).

Note that the state-space tree is dynamically generated based on the states that are visited and expanded according to Algorithm 3. The tree expansion stops once the first goal state is reached, and a near-optimal allocation is found.

Example 2: Figure 3 shows the state-space tree[3] generated by our algorithm for the scenario described in Example 1. A total of 6 states are explored to obtain a complete solution. Starting from the *root state*, we generate the states in the first level of the tree, that is, the states corresponding to task t_1. The *getSuccessors* function discards hosts h_1 and h_3 because they violates the vulnerability and the capacity constraint respectively. Hence, states (t_1, h_2) and (t_1, h_4) are considered and pushed into OPEN along with their f_{vul} values – 0.7 and 1.0 respectively. State (t_1, h_2) is then extracted from OPEN – it is the state with minimum f_{vul} – and considered as the current state. Only one child state is generated by *getSuccessors*, namely, $s^* = (t_2, h_4)$, with $f_{vul}(s^*) = 0.7$. In fact, $g_{vul}(s^*) = g_{vul}(t_1, h_2) + V_{h_4} + \Delta V_{t_2,h_4} = 0.2 + 0.2 + 0 = 0.4$, and $h_{vul}(s^*) = 0.3$. Then state $s^* = (t_2, h_4)$ is pushed into OPEN along with its h_{vul} value. Since state (t_2, h_4) is the entry with the lowest f_{vul} value in OPEN, it is chosen as the current state and its successors (t_3, h_1), (t_3, h_2) and (t_3, h_3) are generated and pushed into OPEN along with their f_{vul} values. State (t_2, h_4) is now pushed into CLOSE. The states corresponding to task t_3 are similarly expanded and visited. The state-space search has now reached a goal state and found a complete task allocation. It pushes state (t_3, h_1) into CLOSE, and returns $a(t_1) = h_2$, $a(t_2) = h_4$ and $a(t_3) = h_1$ as the complete allocation solution. Figure 4 shows the state-space tree when no heuristic is used to estimate h_{vul}, and uniform cost is assumed, i.e., $(\forall s)\, h_{vul}(s) = 0$. In this case, 9 states are generated before finding the same complete task allocation obtained using the heuristic. However, using the heuristic, the entire sub-tree starting at (t_1, h_4) is not expanded, thus improving the performance of state-space search.

[3]Circled numbers indicate the order in which states are generated.

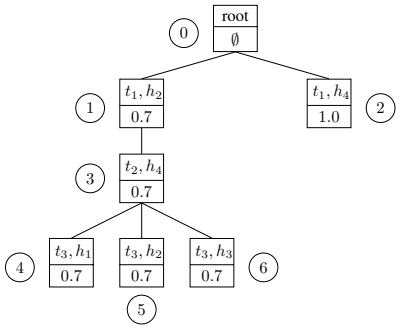

Fig. 3. State-space tree based on the $estimateCost$ heuristic

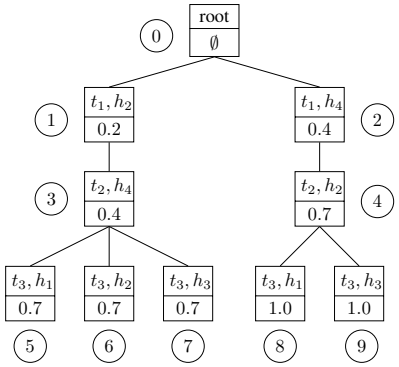

Fig. 4. State-space tree when uniform cost is assumed $h_{vul}(s)$=0

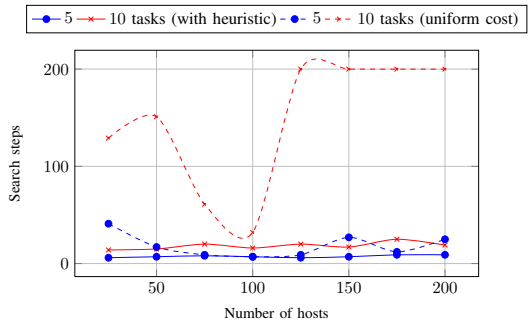

Fig. 5. Number of search steps vs. number of hosts

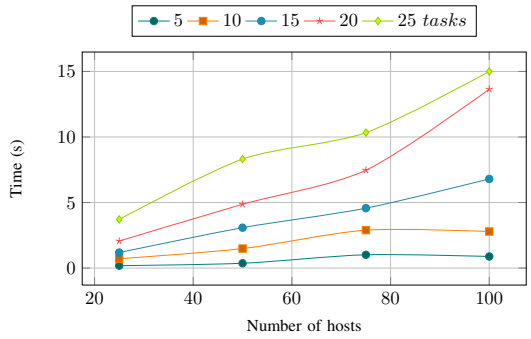

Fig. 6. Processing time vs. number of hosts for different mission sizes

VI. Experimental Results

In this section, we report the experiments we conducted to validate our approach. Our objective is to evaluate the performance of our state-space search algorithm in terms of processing time, number of steps required to identify a complete solution, and approximation ratio for different network configurations and mission scenarios. In order to consider different network configurations, we have randomly initialized the vulnerability score and the available capacity of each host $h_j \in \mathcal{H}$. The vulnerability tolerance and the resource requirements have been similarly initialized for each task $t_i \in T$. All the results reported here are averaged over multiple executions.

First, we show that, as expected, the number of search steps required for complete task allocation using our heuristic is smaller than in the case of uniform cost search, particularly for larger missions. Figure 5 shows how search complexity increases when the number of hosts increases, for $|T| = 5$ and $|T| = 10$. Note that the number of search steps for uniform cost search is comparable to the heuristics-driven search when the number of tasks is 5. However, search complexity increases exponentially as the size of the mission increases.

We have studied the scalability of our algorithm, in terms of processing time, for different network and mission sizes. Figure 6 shows how processing time increases as the number of hosts increases. It is clear that processing time remains in the order of a few milliseconds for small missions ($|T| = 5$ or $|T| = 10$), irrespective of the number of hosts. However, as the size of the mission increases, processing time increases steeply

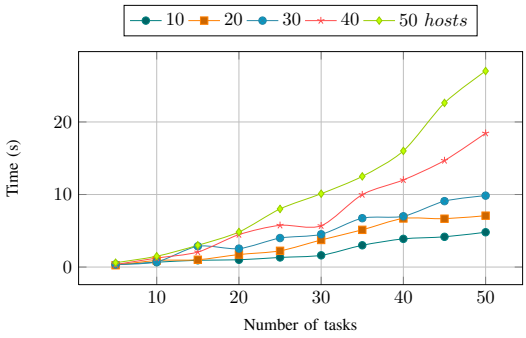

Fig. 7. Processing time vs. number of tasks for different network sizes

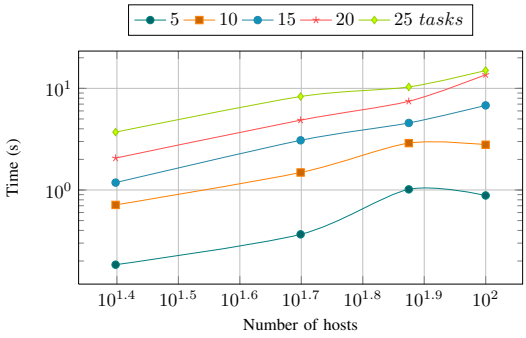

Fig. 8. Processing time vs. number of hosts for different mission sizes

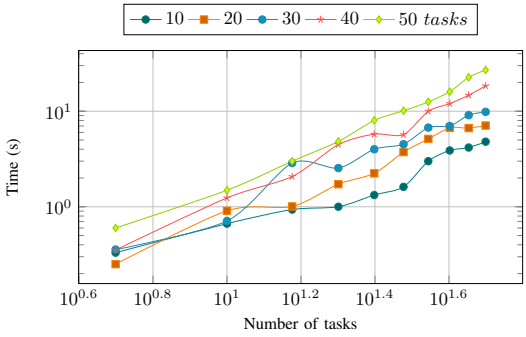

Fig. 9. Processing time vs. number of tasks for different network sizes

(up to 15 seconds for mission with 25 tasks on a network with 100 hosts). Figure 7 shows how processing time increases as the number of tasks increases. Processing times for missions with $10-30$ tasks are roughly similar independently of the size of the network, and increases for missions with $40-50$ tasks (particularly, on networks with over 30 hosts). Figures 8 and 9 show the relationship between processing time, mission size and network size on a logarithmic scale. These charts show that the processing time of our algorithm increases linearly with the size of both missions and networks. Relatively small missions can be allocated on large-scale networks within a few seconds.

Finally, we evaluated the optimality of the solutions obtained by our algorithm from two different perspectives: (i) the mission's exposure to network vulnerabilities and (ii) the quality of deployment in the whole network. Figure 10

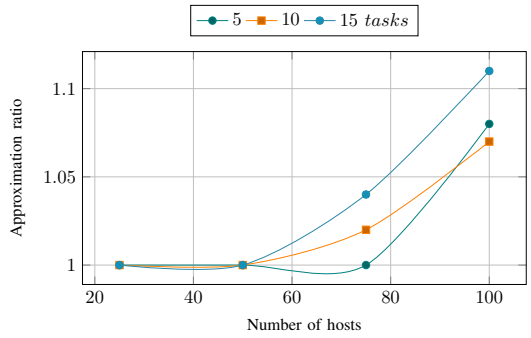

Fig. 10. Approximation ratio vs. number of hosts for different mission sizes, measuring mission's exposure to vulnerabilities

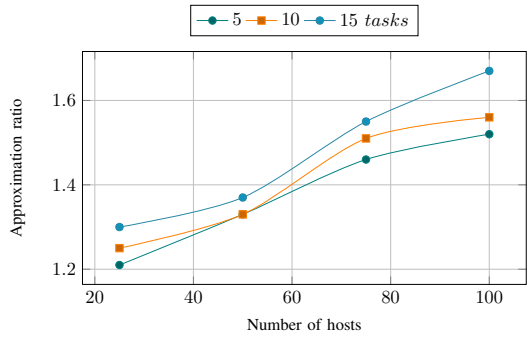

Fig. 11. Approximation ratio vs. number of hosts for different mission sizes, measuring global optimal in the network

shows how the approximation ratio (i.e., the ratio between the aggregated vulnerability score of the mission deployed using our algorithm and that computed using an exhaustive search) changes for different configurations of the network and different mission sizes. It is clear that the proposed approach deploys missions with acceptable security, in a time efficient manner. The approximation ratio remains well within 1.2 in all instances. Similarly, Figure 11 shows how the approximation ratio varies with respect to security across the whole network. In this respect, the variance is low for the missions of all sizes when the network consists of less than 50 hosts.

VII. Conclusions

In this paper, we have highlighted that existing mission deployment solutions do not consider the complex interdependencies between network elements and mission tasks, and, as a consequence, the system remains vulnerable to a wide range of cyber-attacks. We then formulated a security-oriented mission deployment problem and presented a solution based on the A^* algorithm for minimizing the mission's exposure to network vulnerabilities. The proposed approach performs a state-space search to find optimum mission deployment solutions; our heuristic-based A^* approach significantly improves performance by achieving near optimal solutions. We showed experimentally that our algorithm scales linearly with the size of both missions and networks, and provides solutions with with good approximation guarantees. Our future work will be driven towards including temporal aspects of the mission tasks in order to develop a comprehensive, security-oriented scheduling strategy. As part of our future work, we will also study techniques to remove the assumptions on availability of virtual machine images that matches exactly a task's requirements.

References

[1] S. Jajodia, S. Noel, P. Kalapa, M. Albanese, and J. Williams, "Cauldron: Mission-centric cyber situational awareness with defense in depth," in *Proceedings of the Military Communications Conference (MILCOM 2011)*, Baltimore, MD, USA, November 2011, pp. 1339–1344.

[2] V. Mehta, C. Bartzis, H. Zhu, E. Clarke, and J. Wing, "Ranking attack graphs," in *Proceedings of the 9th International Symposium On Recent Advances In Intrusion Detection (RAID 2006)*, ser. Lecture Notes in Computer Science, vol. 4219, Hamburg, Germany, September 2006, pp. 127–144.

[3] G. Jakobson, "Mission cyber security situation assessment using impact dependency graphs," in *Proceedings of the 14th International Conference on Information Fusion (FUSION)*, Chicago, IL, USA, July 2011.

[4] M. Albanese, S. Jajodia, and S. Noel, "Time-efficient and cost-effective network hardening using attack graphs," in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, Boston, MA, USA, June 2012.

[5] R. Jhawar, V. Piuri, and P. Samarati, "Supporting security requirements for resource management in cloud computing," in *Proceedings of the 15th IEEE International Conference on Computational Science and Engineering (CSE 2012)*, Paphos, Cyprus, December 2012, pp. 170–177.

[6] T. Xie and X. Qin, "Performance evaluation of a new scheduling algorithm for distributed systems with security heterogeneity," *Journal of Parallel and Distributed Computing*, vol. 67, no. 10, pp. 1067–1081, October 2007.

[7] S.-W. Xiong, Y.-X. Zhao, and N. Xu, "SAREC-GA: A security-aware real-time scheduling algorithm with genetic algorithm," in *Proceedings of the 6th International Conference on Machine Learning and Cybernetics (ICMLC 2007)*, vol. 6, Hong Kong, August 2007, pp. 3122–3127.

[8] S. Srinivasan and N. K. Jha, "Safety and reliability driven task allocation in distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 3, pp. 238–251, March 1999.

[9] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, April 1977.

[10] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions," in *Proceedings of the 7th Heterogeneous Computing Workshop (HCW 1998)*, Orlando, FL, USA, March 1998, pp. 79–87.

[11] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1384–1397, November 1988.

[12] X. Zhu, C. Santos, D. Beyer, J. Ward, and S. Singhal, "Automated application component placement in data centers using mathematical programming," *International Journal of Network Management*, vol. 18, no. 6, pp. 467–483, November 2008.

[13] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

[14] C.-C. Shen and W.-H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Transactions on Computers*, vol. 34, no. 3, pp. 197–203, March 1985.

[15] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 42–50, July 1998.