Securing Mission-Centric Operations in the Cloud[*]

Massimiliano Albanese, Sushil Jajodia, Ravi Jhawar, and Vincenzo Piuri

Abstract Recent years have seen a growing interest in the use of Cloud Computing facilities to execute critical missions. However, due to their inherent complexity, most Cloud Computing services are vulnerable to multiple types of cyber-attacks and prone to a number of failures. Current solutions focus either on the infrastructure itself or on mission analysis, but fail to consider the complex interdependencies between system components, vulnerabilities, failures, and mission tasks. In this chapter, we propose a different approach, and present a solution for deploying missions in the cloud 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, and propose a solution based on the A^* algorithm for searching the solution space. Additionally, in order to provide missions with further availability and fault tolerance guarantees, we propose a cost-effective approach to harden the set of computational resources that have been selected for executing a given mission. Finally, we consider offering fault tolerance as a service to users in need of deploying missions in the Cloud. This approach allows missions to obtain required fault tolerance guarantees from a third party in a transparent manner.

Massimiliano Albanese and Sushil Jajodia
Center for Secure Information Systems, George Mason University, Fairfax, VA, USA
e-mail: {malbanes, jajodia}@gmu.edu

Ravi Jhawar and Vincenzo Piuri
Department of Computer Science, Università degli Studi di Milano, Crema, Italy
e-mail: {ravi.jhawar,vincenzo.piuri}@unimi.it

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

1 Introduction

In recent years, individuals and organizations are increasingly resorting to Cloud-based services for storage, processing, and management of their data and applications. This practice offers several advantages to application and data owners – users, in general – with respect to traditional in-house management. First, users are relieved from buying expensive hardware and software licenses, and recruiting skilled personnel to administer and maintain their computing resources, thus providing significant economic savings. Second, users can access their applications using any device providing Internet connectivity. Third, even individuals with little or no IT background can take advantage of Cloud-based services to develop applications with very high scalability and elasticity requirements. These benefits are also providing an incentive for users to leverage Cloud-based solutions to deploy mission-critical applications.

A Cloud computing infrastructure is typically built by inter-connecting massive amounts of hardware according to well-defined design patterns, resulting in large-scale data centers that can elastically deliver computing resources to the users through virtualization. The main problem of adopting such Cloud-based Infrastructure-as-a-Service (IaaS) model is that the data centers, due to their very high complexity, may be vulnerable to various cyber-attacks and subject to a large number of failures, which are not within the control scope of the users, thus increasing users's security and fault tolerance concerns [1]. We identify two primary reasons why state-of-art techniques are unable to suitably address such concerns:

- Most security solutions either design data centers integrating tools such as intrusion detection systems and firewalls, or develop strategies to implement applications using techniques such as data obfuscation and memory management. However, interdependencies between the infrastructure, applications, and residual vulnerabilities are not taken into account.
- Fault tolerance methods are generally applied at application procurement and development time. This approach requires users to build their applications by considering environment specific parameters. However, it is infeasible to combine failure behavior and system architecture in Cloud computing due to the limited information about the infrastructure that providers release to the users.

The goal of this chapter is to provide an overview of approaches that can address the aforementioned problems. We discuss the inherent challenges, possible solutions, and relevant open issues. In particular, as an approach to address the security issues, we describe a solution that considers the current vulnerability status of the infrastructure and deploys mission-critical applications (or simply, *missions*) so as to minimize their exposure to existing network vulnerabilities. Once a mission is deployed, the proposed solution then protects the resources (computational hosts and network links) used by the mission to ensure high levels of security during mission execution (Section 3 and Section 4). Note that this approach is mission-centric and aims at providing maximum security for missions, given the current state of the infrastructure. To address the fault tolerance issues, we discuss a scheme that can help design fault tolerance solutions based on users' requirements at runtime and apply it to missions in a transparent manner. The latter approach can be integrated within the overall framework for delivering fault tolerance as a service to users' applications or missions (Section 5).

2 Background

In this section, we present some preliminary concepts and our assumptions about the Cloud infrastructure and the missions. We discuss the vulnerability behavior and failure characteristics of typical Cloud infrastructures, and the requirements for satisfying a mission's dependability goals.

2.1 Cloud infrastructure

A Cloud computing infrastructure is typically built by inter-connecting large-scale, geographically distributed, data centers. Each data center consists of thousands of hosts that are organized into racks and clusters, and each host contains multiple processors, storage disks, memory modules and network interfaces. Physical hosts are first connected via high-speed rack switches, which are in turn connected to aggregation switches (AggS), forming a subsystem that can be viewed as a cluster. A cluster groups hosts with similar resource characteristics or administrative parameters. An AggS connects tens of racks to redundant access routers (AccR) that finally connects different data centers via the Internet backbone. Typically, data centers also deploy security services (e.g., firewalls, intrusion detection systems) to protect network elements from potential threats, and install hypervisors on physical hosts so that VMs with desired size and software stack can be instantiated and delivered to the users upon request.

Vulnerability characteristics. Despite careful security engineering, a number of vulnerabilities remain in the network and allow malicious adversaries to launch different types of cyber-attacks. For example, an attacker may exploit vulnerabilities in services such as *ftp*, *rsh*, and *sshd* to gain desired access privileges on a given host. Such exploits can be used to compromise users' applications or missions deployed in the system. Vulnerabilities and attack paths in the network can be analyzed using vulnerability scanners, and approaches based on attack graphs, dependency graphs, and attack surfaces (e.g., [2, 3, 4, 5]). Analysis tools can also be extended with probabilistic schemes and ranking methods to quantify the vulnerability level of individual hosts. For simplicity, in this chapter, we assume that a vulnerability value V_h is pre-computed for each host $h \in \mathcal{H}$ in the infrastructure by adopting one of the existing techniques.

A physical host $h \in \mathcal{H}$ in the infrastructure can be characterized using a vector $\overrightarrow{h} = (h[1], h[2], \ldots, h[d], h[d+1])$, where the first d dimensions represent the host's residual capacity for each resource type (e.g., CPU, memory). The $d + 1^{th}$ dimension represents the host's vulnerability value V_h. The residual resource capacities and the vulnerability level of each host are represented using normalized values in $[0, 1]$. For example, $\overrightarrow{h} = (cpu, mem, V_h) = (1, 1, 1)$, where $cpu = 1$ and $mem = 1$, implies that both resources are fully available, whereas $V_h = 1$ means that the host is extremely vulnerable.

Failure behavior. Due to their high complexity, infrastructure components are subject to a large number failures that may prevent the system from fulfilling its intended functionality. Research on a system's failure characteristics is necessary because infrastructure failures may have a significant impact on the applications deployed in the Cloud. Several researchers54 [6, 7] used data mining techniques to understand the failure behavior of data center components. Examples of key observations from these studies are as follows:

- The annual failure rate for servers is around 8%. The average number of repairs is 2 per machine (e.g., 20 repair or replacement events in 9 machines were identified over a 14 months period).
- Hard disks are the most failure-prone hardware components and the most significant reason behind server failures (about 78% of total faults or replacements affected hard disks).
- Among network devices, top-of-rack switches are most reliable (failure rate less than 5%) and load balancers are least reliable (failure probability of 1 in 5). Load balancers mainly fail due to software bugs and configuration errors, and experience short but frequent failures.

The failure behavior of various server and network components can also be analyzed using analytical models such as fault trees and Markov chains [8, 9]. Such modeling techniques, as discussed in Section 5, can be used to analyze the impact of component failures on users' applications.

2.2 Missions

We consider a mission M to be a composition of a set of tasks $M = \{\tau_1, \ldots, \tau_m\}$. This model-independent definition allows us to consider different software architectures for the mission (e.g., web services, business processes, scientific applications) as well as different formalisms (e.g., Petri Nets, work flows). For example, a mission can be a three-tier web application realizing an e-Commerce service or a scientific tool with tasks performing graph theoretical calculations on geographical maps. Intuitively, a mission is successful if (i) all the tasks start from a correct initial state, perform their operations, and generate the correct output in a specified amount of time, and (ii) the protocol that composes the information from individual tasks can justifiably be trusted. Each task in the mission can be associated with a tolerance value *tol* when it is implemented using some security mechanisms (e.g., memory management guards to protect from buffer overflow attacks). Intuitively, the *tol* value provides an estimate of the maximum level of vulnerability that the task can be exposed to without compromising its successful completion. Each mission task may also be replicated to tolerate failures. In fact, we create a set of task replicas $R_k = \{\tau_k^1, \ldots, \tau_k^{|R_k|}\}$ for each task, and the overall mission becomes a composition of the set of replicated task sets $T = \{t_i\} = \bigcup_{\tau_k \in M} R_k$. We treat task replicas as independent tasks for the purpose of mission deployment.

Similarly to physical hosts, we characterize each mission task using 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 (e.g., CPU, memory) and the $d+1^{th}$ dimension is the task's maximum vulnerability tolerance value tol. Resource requirements and vulnerability tolerance are also represented using normalized values in $[0,1]$, e.g., $\overrightarrow{t} = (cpu, mem, tol) = (0.5, 0.6, 0.6)$.

Security of the mission. A number of aspects must be considered to securely operate a given mission in a Cloud infrastructure. In this chapter, instead of considering traditional approaches based on network hardening or applying software security techniques, we study the following aspects:

- *Secure mission deployment*: Given a mission and the current vulnerability state of the infrastructure, deploy the mission's tasks in the network using the resources (hosts and network links) that are most suitable for successfully executing the mission. We formulate this problem as a task allocation problem that minimizes the mission's exposure to existing vulnerabilities. We consider both dynamic and static versions of this problem by modeling missions ignoring or considering temporal aspects respectively (see Section 3).
- *Static and dynamic resource protection*: Given a mission and the resources it uses (after it has been deployed), harden these resources in a way that is optimal with respect to a given cost function, in order to ensure high levels of security to the mission during execution. The static version of the problem protects resources for the entire duration of the execution whereas the dynamic version protects only the resources still to be used for execution (see Section 4).

Both these aspects should be addressed for any given mission in order to ensure that it achieves high levels of security in the Cloud. Note that the application of the resource protection scheme does not change the solution space of the mission deployment scheme. Therefore, the above two aspects generate independent, yet complementary, results that together allow the mission's execution in the Cloud infrastructure in a way that minimizes its exposure to vulnerabilities and the impact of exploits.

Fault tolerance of the mission. Implementing a fault tolerant mission using traditional approaches may be infeasible since the system's architectural details are not widely available to Cloud computing users. As a consequence, a new approach to address fault tolerance issues of missions is necessary. In this chapter, we discuss an approach where missions can obtain required fault tolerance properties *as a service* from a third-party *fault tolerance service provider*. In particular, we study the following aspect:

- *Fault tolerance management*: Given a mission and its fault tolerance requirements, apply a comprehensive fault tolerance solution to the mission and ascertain users' requirements at runtime. We present a scheme that realizes general fault tolerance mechanisms as independent modules that can transparently function on the missions, and based on user's requirements, appropriate modules are selected and composed in a specified manner to form a comprehensive solution (see Section 5).

3 Secure Mission Deployment

The first step to securely execute a given mission is to deploy the mission tasks in the Cloud such that their exposure to vulnerabilities is minimized. Since requests for mission deployment may arrive at any time, we develop a deployment strategy that considers the current resource allocation and vulnerability status of the Cloud. When a request is received, the allocation for the new mission is computed based on the availability of currently unused resources. Once the mission is deployed, resource allocation and vulnerability status are updated accordingly. In this section, we present a detailed problem formulation, and an approach to solve the mission deployment problem. We also discuss the challenges that still need to be addressed.

Problem formulation. To focus on the deployment problem, we assume that a virtual machine containing required resources and services is instantiated for each task in the mission. This assumption reduces mission deployment to a task allocation problem that can be characterized as a function $a : T \to \mathcal{H}$ which maps each mission task $t_i \in T$ to a physical host $h_j \in \mathcal{H}$ in the infrastructure. The binary variable a_{ij} denotes the truth value of $a(t_i) = h_j$, that is,

$$(\forall t_i \in T, h_j \in \mathcal{H}) \quad a_{ij} = \begin{cases} 1 & \text{if } a(t_i) = h_j \\ 0 & \text{otherwise} \end{cases}$$

Every time a task t_i is allocated on host h_j, the vulnerability score of host h_j may increase by $\Delta V_{t_i,h_j}$ since new vulnerabilities are potentially introduced on the host. Note that, although multiple hosts may have similar configurations and, consequently, similar vulnerability scores, their vulnerability scores may vary significantly at run time, 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

$$\min_{a \in \mathcal{A}} \max_{h_j \in \mathcal{H} | \exists t_i \in T, a(t_i) = h_j} V^*_{h_j} \tag{1}$$

Note that, ideally, the mission's exposure to vulnerabilities in the system after allocation should be zero. In practice, the effectiveness of task allocation must be measured in terms of the deviation from the ideal behavior. Furthermore, note that this formulation focuses on optimizing the security and workload of the mission. The fault tolerance aspects of the mission are integrated in the optimization problem in the form of constraints on the placement of each mission task in the Cloud infrastructure (e.g., the distribution constraint described below). We provide a detailed discussion on fault tolerance constraints, and a method to derive them, in Section 5.

Each allocation $a \in \mathcal{A}$ should satisfy the following constraints to ensure the dependability of the mission.

- *Consistent allocation*: This constraint specifies two conditions that must be satisfied across all the hosts in the infrastructure at all times.

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

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

  Equation 2 specifies that each mission task must be allocated only on a single physical host. Equation 3 implies that the amount of resources consumed by all the tasks mapped on a single host cannot exceed the total capacity of that host in any dimension.
- *Distribution*: Equation 4 specifies that the allocation function $a : T \to \mathcal{H}$ must map all the replicas of a task on different hosts to avoid single points of failure.

$$(\forall \tau_k \in M)(\forall \tau'_k, \tau''_k \in R_k) \quad a(\tau'_k) \neq a(\tau''_k) \tag{4}$$

- *Vulnerability tolerance*: To protect the tasks from being compromised due to the vulnerabilities on the hosts on which they are allocated, this constraint specifies that a task can be mapped only to the hosts whose vulnerability value V is less than the vulnerability tolerance *tol* of that task, that is,

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

An attacker can exploit the vulnerabilities on a given host h_j and compromise the mission if a task $t_i \in T$ is placed on host h_j having vulnerability value higher than the tolerance level of the task.

Mission deployment solution. Modeling secure mission deployment as an optimization problem has not been well-studied in the literature. Given the NP-hardness of the general allocation problem, existing solutions typically adopt heuristics, meta-heuristics, and mathematical programming based approaches. In general, such approaches either have scalability issues or relax the optimality goals. In our context, we need an approach that solves the mission deployment problem in a time-efficient manner and provides acceptable sub-optimal results. One possible solution is based on the A^* state-space search method discussed in [10]. Here, we provide a detailed description of this approach.

To enable A^* exploration, the overall state-space is represented as a tree. We start by describing the data structure supporting the exploration of the solution using the A^* algorithm:

- A *state* s is a possible choice for allocating task t_i on host h_j. A state is represented by the pair (t_i, h_j).
- The *root state* represents is the initial state from which the algorithm starts, with no task being allocated yet.
- An *operation* of the A^* algorithm generates the set of feasible child states for a given state s.
- The *solution path* is the path from the root state to the first leaf state that is reached during state-space exploration.
- The *goal state* is a state in which all the tasks have been allocated. A leaf state corresponds to a complete allocation.

To generate the search tree from the root state, the set T of tasks is initially sorted in increasing order of vulnerability tolerance *tol* and considered for allocation in this order. The i^{th} task in the sorted list corresponds to the i^{th} level in the state-space tree. Given a state $s = (t_i, h_j)$, the next task t' from the sorted list is chosen, and all the hosts $h_j \in \mathcal{H}$ that satisfy the dependability constraints (consistent allocation, distribution, and vulnerability tolerance constraints) with respect to t' are shortlisted. The successors of state s are all the states mapping t' to one of the shortlisted hosts.

The evaluation function for state s in the state-space tree is as follows:

$$f_{vul}(s) = g_{vul}(s) + h_{vul}(s) \tag{6}$$

where $g_{vul}(s)$ is the aggregate vulnerability score associated with the allocation path from the root state to the current state s, and $h_{vul}(s)$ estimates

Algorithm 1 Estimate cost

```
1: Repeat steps 2 through 4 until a goal state is reached.
2: Use the $A^*$ operation to obtain the set $S$ of feasible successors of the current state.
3: Calculate $(V_{h_j} + \mathit{\Delta} V_{t_i,h_j})$ for each state in $S$.
4: Select the state with minimum $(V_{h_j} + \mathit{\Delta} V_{t_i,h_j})$ value and temporarily mark it as the
   current state $s$. {Note that we choose the state with minimum value to keep the value
   of $h_{vul}(s)$ as the lower bound.}
```

Algorithm 2 State-space tree traversal scheme

```
1: Push the root state in OPEN and execute steps 2–5 until either a complete allocation is
   obtained or OPEN becomes empty.
2: Pop the state $s$ with minimum $f_{vul}(s)$ from OPEN.
3: If state $s$ corresponds to the goal state, construct the final solution by traversing the tree
   in the reverse order from the goal state to the root state; else, generate the successors
   of $s$ using the $A^*$ operation.
4: For each successor $s^*$ of $s$
     i) Calculate $new\_g_{vul}$, the aggregate vulnerability from the root state to state $s^*$.
    ii) If the 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 since the same
        state has already been reached with lower cost. Otherwise, continue with the next
        step.
   iii) Estimate the lower bound $h_{vul}(s^*)$ and compute $f_{vul}(s^*) = g_{vul}(s^*) + h_{vul}(s^*)$.
    iv) Push the successor $s^*$ and its $f_{vul}(s^*)$ value in OPEN since state $s^*$ has been suc-
        cessfully generated and its cost computed.
5: Push the parent state $s$ in CLOSE since it has been visited.
```

the minimum additional vulnerability associated with completing the allocation from state s to a goal state. The value of $g_{vul}(s)$ is computed as follows:

$$g_{vul}(s) = g_{vul}(parent(s)) + V_{h_j} + \mathit{\Delta} V_{t_i,h_j} \tag{7}$$

where $g_{vul}(parent(s))$ denotes the aggregate vulnerability score associated with the allocation path leading to the parent state of s and $(V_{h_j} + \mathit{\Delta} V_{t_i,h_j})$ denotes the updated vulnerability score of host h_j after allocation of task t_i. The $gvul(s)$ value for the root state is initialized to 0.

If we consider a uniform cost search, the lower bound estimate for each state must be considered zero, that is, $h_{vul}(s) = 0$. The A^* algorithm, in this case, obtains an optimal solution but expands a higher number of states (as shown in Example 1). Therefore, a heuristic is necessary to estimate h_{vul}. Algorithm 1 outlines an approach to realize the *estimateCost* function. In this case, h_{vul} is computed as the total vulnerability value along the traversed path. This algorithm significantly improves the performance of the traversal scheme when compared to a uniform cost search without influencing the final result.

The state-space tree traversal scheme provides the solution path that minimizes the mission's exposure to the vulnerabilities in the system. The traversal scheme dynamically generates the state-space tree based on the states that are expanded and visited. The tree expansion starts from the root state and stops at the goal state, where it obtains a near-optimal allocation. Two data structures `OPEN` and `CLOSE` are used for making the traversal decisions. `OPEN` contains the set of states that are generated using the A^* *operation* but not yet visited, and `CLOSE` contains the states that are already visited. Each entry in `OPEN` and `CLOSE` contains a state s and its corresponding $f_{vul}(s)$ value. Algorithm 2 outlines the traversal scheme using (i) the A^* *operation* which, given a state s, generates the set of feasible child states or successors, and (ii) the *estimateCost* heuristic that calculates the lower bound vulnerability value h_{vul} of each successor.

Table 1 Example scenario for mission deployment

Infrastructure		**Mission**	
Host	**Residual CPU capacity, Vulnerability level**	**Task**	**CPU Requirement, Vulnerability tolerance**
$h_j \in \mathcal{H}$	$\overrightarrow{h}(cpu, V)$	$t_i \in T$	$\overrightarrow{t}(cpu, tol)$
h_1	0.5, 0.2	t_1	0.4, 0.2
h_2	0.3, 0.2	t_2	0.4, 0.2
h_3	0.7, 0.1	t_3	0.3, 0.4
h_4	0.5, 0.3		

Table 2 Increase in vulnerability scores

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

Example 1. Consider an infrastructure with four hosts $\mathcal{H} = \{h_1, \dots, h_4\}$ and a mission with two tasks $M = \{\tau_1, \tau_2\}$, where $\mathcal{R}_1 = \{\tau_1^1, \tau_1^2\}$ and $\mathcal{R}_2 = \{\tau_2^1\}$. Mission deployment is driven by $a : \{t_1, t_2, t_3\} \to \{h_1, h_2, h_3, h_4\}$, and distribute constraint holds for tasks t_1 and t_2. For simplicity, consider only a single resource dimension for hosts and tasks (say CPU). Table 1 outlines available CPU capacity and vulnerability level of each host, and CPU requirements and vulnerability tolerance threshold of each task. Table 2 provides details on the increase in the vulnerability scores.

Figure 1(a) illustrates the state-space tree generated by our algorithm during mission deployment. The algorithm starts from the *root state* by generating the states for the first level in the tree. The *operation* considers task t_1, discards hosts h_2 and h_4 since they violate the capacity and vulnerability threshold constraints respectively, and generates states (t_1, h_3) and (t_1, h_1).

The $f_{vul}(s)$ values for the two states are calculated as 0.7 and 1.0 respectively and pushed into OPEN.

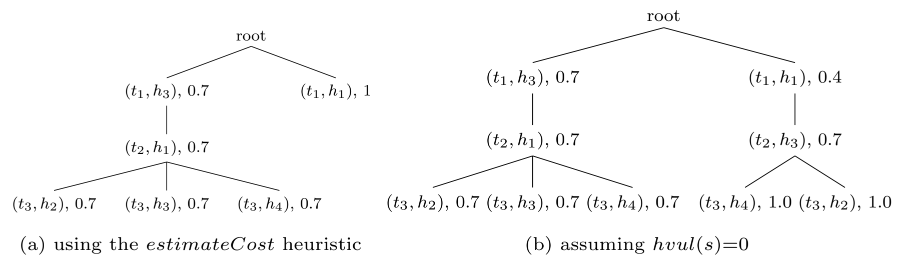

Fig. 1 State-space tree expanded using the A^* traversal scheme

Since state (t_1, h_3) has the smallest $f_{vul}(s)$ value, it is extracted from OPEN and marked as the current state. Its successors are then generated and f_{vul} values calculated. In this case only state $s = (t_2, h_1)$ with $fvul(s) = 0.7$ is returned and pushed into OPEN. In particular, after calculating $g_{vul}(s) = 0.4$, the *estimateCost* function is used to estimate the vulnerability value $h_{vul}(s)$ along this path. In this case, feasible states corresponding to task t_3 are considered, and the state with minimum $g_{vul}(s)$ value (0.3) is returned since states corresponding to task t_3 are leaf nodes.

At this point, state (t_2, h_1) is the entry with the lowest $f_{vul}(s)$ value in OPEN. This state is marked as the current state and its successors (t_3, h_4), (t_3, h_3) and (t_3, h_2) are generated. The $f_{vul}(s)$ value of all these states are calculated and pushed in OPEN. The state (t_2, h_1) is now pushed in CLOSE. The states corresponding to the task t_3 are similarly expanded and visited. The state-space search has now reached the goal state and found the complete task allocation. The algorithm pushes (t_3, h_4) in CLOSE, and returns $a(t_1)$=h_3, $a(t_2)$=h_1 and $a(t_3)$=h_4 as the complete allocation solution.

When uniform cost is assumed (i.e., $(\forall s)\, h_{vul}(s) = 0$) and this heuristic is not used, 9 states are expanded to perform task allocation, as shown in Figure 1(b), while our algorithm expands only 6 states.

Open issues. Based on the above formulation of the mission deployment problem, we identify that three main challenges still need to be addressed.

- *VM images selection*: For each task, we need to instantiate a virtual machine containing all the resources and the services required to successfully execute that task. Hence, during mission deployment, we must first map each task to an available VM image and then to a physical host.
  Existing Cloud computing services usually require users to manually select VM images from a repository. Typically, users can also upload and share their VM images with other customers. This feature exacerbates the security problems in public Cloud services, and such problems cannot be

identified by the users in a straightforward manner during image selection. For example, Balduzzi et al. [11] studied the vulnerability issues in Amazon EC2 service[2] by analyzing over 5,000 public images; using the Nessus vulnerability scanner, they identified that 98% of Windows AMIs (Amazon Machine Images) and 58% of Linux AMIs had software with critical vulnerabilities. This implies that an automated security-driven search scheme is required to deploy mission tasks. In other words, a task allocation function $a^{image} : T \rightarrow \mathcal{I}$ which maps each task $t \in T$ to a VM image $I \in \mathcal{I}$ based on security requirements needs to be defined.

- *Dynamic mission deployment*: Instead of allocating resources to tasks for the entire duration of a mission, we must consider the execution time of each task and perform allocation only for necessary periods of time while minimizing its exposure to the vulnerabilities.
  The mission model must be extended to include the start time and a deadline for each task. This extension allows us to generate the target execution timeline of the mission and obtain an enhanced mission model. This mission model is a special kind of labeled graph $M = (S, T, \rho)$, where S is a set of nodes representing the state of the computation, T is a set of edges representing tasks, and $\rho : T \rightarrow 2^R$ is a function mapping each task to the pool of resource types required to complete the task. Additionally, edges are labeled with task durations. Similar to the mission deployment approach discussed in this section, scalable solutions that can efficiently schedule mission tasks are required to address this challenge.
- *Incremental vulnerability analysis*: Each allocation introduces a set of new services on a host and increases its vulnerability level. We need a function $v : \mathcal{H} \times T \rightarrow \mathbb{R}$ that can estimate the increase in the vulnerability level $\Delta V_{t_i,h_j}$ to facilitate the "what-if" analysis. One possible approach to vulnerability assessment is by means of attack graphs, and a naive method to estimate $\Delta V_{t_i,h_j}$ is to discard the original attack graph and perform re-computation from scratch using the new data. However, such re-computation is wasteful since typically the changes are small, resulting in information that is not very different from the original one. Therefore, we need to take an incremental approach that (i) identifies the portions of the attack graph that have changed due to an event, (ii) re-computes the vulnerability information only in the changed portion, and (iii) combines the new and original information to provide updated results.

4 Mission Protection

The second step to securely execute a mission is to protect the hosts and network links used by the mission from possible cyber-attacks. In this section,

[2] http://aws.amazon.com/ec2/

we formulate the hardening problem and the cost model, and discuss the approach presented in [2] to solve the problem using attack graphs.

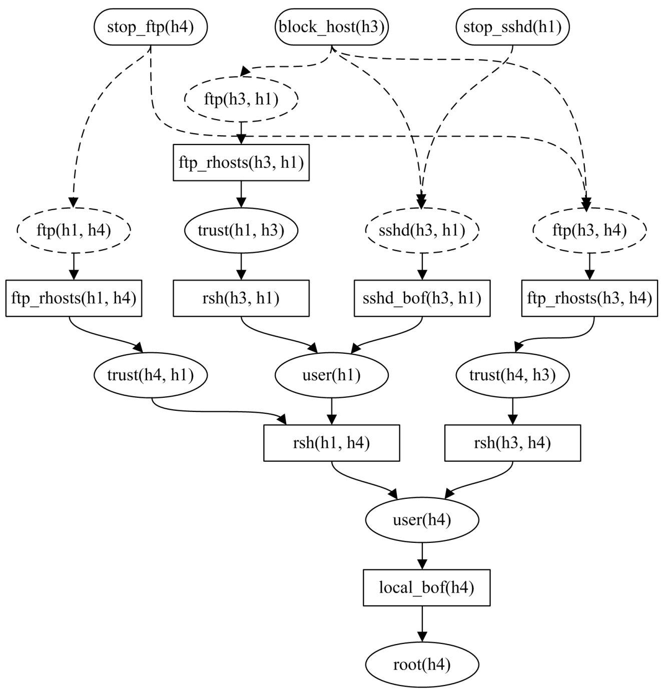

Fig. 2 Example of an attack graph including possible hardening actions, initial conditions, intermediate conditions, and exploits

Problem formulation. A network hardening strategy is a set of atomic actions that can be taken to guard various resources in the network. For instance, an action may consist in stopping the *ftp* service on a given host. We start by introducing the notion of attack graphs that represent prior knowledge about vulnerabilities, their dependencies, and network connectivity. Given a set E of exploits, a set of security conditions C (e.g., existence of a vulnerability on a host or connectivity between two hosts), a require relation $R_r \subseteq C \times E$, and an imply relation $R_r \subseteq E \times C$, an *attack graph* is a directed graph $G = (E \cup C, R_r \cup R_i)$, where $E \cup C$ is the vertex set and $R_r \cup R_i$ is the edge set [2]. The term *Initial conditions* refers to the subset of conditions $C_i = \{c \in C \mid \nexists e \in E \text{ s.t. } (e, c) \in R_i\}$, whereas other conditions, which are usually consequences of exploits, are referred to as intermediate conditions.

Example 2. In Example 1, mission tasks are allocated on hosts h_3, h_1 and h_4. Assume that our objective is to prevent the attacker from gaining root privileges on host h_4, i.e., we want to avoid reaching condition $root(h_4)$ so as to protect task t_3.

Figure 2 illustrates an example attack graph in which exploits are represented using rectangles and conditions using ovals. The dashed ovals are the initial conditions and other ovals represent intermediate conditions. The attack graph is simplified in several ways. For example, a single condition $ftp(h_s, h_d)$ is used to denote transport-layer *ftp* connectivity between two hosts h_s and h_d, physical-layer connectivity, and existence of the *ftp* daemon on host h_d. The attack graph depicts a simple scenario, with hosts h_3, h_1 and h_4, and four types of vulnerabilities: *ftp_rhosts*, *rsh*, *sshd_bof*, and *local_bof*. An example of attack path is the one where the attacker starts by establishing a trust relationship with host h_4 (condition $trust(h_4, h_3)$) by exploiting an *ftp* vulnerability on host h_4 ($ftp\_rhosts(h_3, h_4)$). The attacker can then gain user privileges on host h_4 (condition $user(h_4)$) with an *rsh* login, and achieve the goal condition $root(h_4)$ using a local buffer overflow attack.

An allowable *hardening action* is any subset of initial conditions such that all the conditions can be jointly disabled in a single step, and no other initial condition is disabled as a consequence. The rounded rectangles in the attack graph in Figure 2 are examples of allowable hardening actions:

- $stop\_ftp(h_4) = \{ftp(h_1, h_4), ftp(h_3, h_4)\}$
- $block\_host(h_3) = \{ftp(h_3, h_1), sshd(h_3, h_1), ftp(h_3, h_4)\}$
- $stop\_sshd(h_1) = \{sshd(h_3, h_1)\}$

Given an attack graph, a set A of allowable actions and a set of target conditions $C_t = \{c_1, \ldots, c_n\}$, a *hardening strategy* S is a set of hardening actions such that conditions in C_t cannot be reached after all the actions in S are applied.

Note that removing specific initial conditions may require to take actions that disable additional conditions (e.g., conditions that are not part of any attack path). Therefore, in order to obtain a cost-effective *hardening strategy*, we need to define a cost model that takes the impact of hardening actions into account. A *hardening cost function* is any function $cost : \mathcal{S} \to \mathbb{R}^+$ that satisfies the following conditions:

$$cost(\emptyset) = 0 \tag{8}$$

$$(\forall S_1, S_2 \in \mathcal{S})(C(S_1) \subseteq C(S_2) \implies cost(S_1) \leq cost(S_2)) \tag{9}$$

$$(\forall S_1, S_2 \in \mathcal{S})(cost(S_1 \cup S_2) \leq cost(S_1) + cost(S_2)) \tag{10}$$

where $\mathcal{S}$ denotes the set of all possible strategies and $C(S)$ denotes the set of all conditions disabled under strategy S. Note that many different cost functions can be defined. For example, a basic cost function could simply count the number of initial conditions that are removed under a hardening strategy. Two possible hardening strategies for the attack graph Figure 2 are $S_1 = \{stop\_ftp(h_4)\}$ and $S_2 = \{block\_host(h_3)\}$. If we assume that $cost(\{stop\_ftp(h_4)\}) = 20$ and $cost(\{block\_host(h_3)\}) = 10$, then the optimal strategy with respect to $root(h_4)$ is $S_2 = \{block\_host(h_3)\}$.

Mission protection solution. Most hardening techniques starts from the target conditions and move backwards through the attack graph to make logical inferences. Such backward search schemes typically face combinatorial explosion issues. Therefore, we must define a scalable scheme to build hardening strategies.

Starting from initial conditions, the hardening scheme in [2] traverses the attack graph forward. A key advantage of traversing the attack graph forward is that in a single pass, the algorithm can compute hardening strategies with respect to any condition. More importantly, forward traversal enables us to prune the search space, as briefly discussed below. The hardening algorithm first performs a topological sort of the nodes in the attack graph, and pushes them into a queue, with initial conditions at the front of the queue. Each node q in the queue is then analyzed and a set $\sigma(q)$ of possible hardening strategies w.r.t. to q is determined. Based on the nature of q (exploit or security condition), different steps are taken to compute $\sigma(q)$, as described in the following.

- If q is an *initial condition*, it is associated with a set of strategies $\sigma(q)$ such that each strategy contains one and only one of the allowable actions that disable q.
- If q is an *exploit*, it is associated with a set of strategies $\sigma(q)$ that is the union of the sets of strategies for each condition c required by q. In fact, an exploit can be prevented by disabling at least one of its required conditions.
- If q is an *intermediate condition*, it is associated with a set of strategies $\sigma(q)$ such that each strategy is the union of a strategy for each of the exploits that imply q. In fact, in order to prevent the attacker from reaching an intermediate condition, all the exploits that imply it must be prevented

In order to prevent the combinatorial explosion of the search space, the algorithm only maintains the k best solution w.r.t. cost for each intermediate node. Setting $k = 1$ will result in very fast execution, but will provide more expensive solutions. Higher values of k will increase execution times but will result in solutions that are closer to the optimal one. This scheme, under reasonable assumptions, provides an approximation ratio that, for $k = 1$, is bounded by $n^{d/2}$, where n is the maximum in-degree of nodes in the graph and d is the depth of the graph. Additionally, experiments reported in [2] show that, in practice, the approximation ratio is much lower than its theoretical upper bound.

Example 3. Consider again the attack graph of Figure 2, and assume that the cost of actions $stop\_ftp(h_4)$, $block\_host(h_3)$, and $stop\_sshd(h_1)$ is 20, 10, and 15 respectively. After executing the topological sort and examining initial conditions, using $k = 1$, we obtain the following intermediate results:

- $\sigma(ftp(h_1, h_4)) = \{\{stop\_ftp(h_4)\}\}$
- $\sigma(ftp(h_3, h_1)) = \{\{block\_host(h_3)\}\}$
- $\sigma(sshd(h_3, h_1)) = \{\{block\_host(h_3)\}\}$
- $\sigma(ftp(h_3, h_4)) = \{\{block\_host(h_3)\}\}$

When the algorithm examines the exploit $rsh(h_1, h_4)$, before pruning we obtain $\sigma(rsh(h_1, h_4)) = \{\{stop\_ftp(h_4)\}, \{block\_host(h_3)\}\}$. After pruning, we obtain $\sigma(rsh(h_1, h_4)) = \{\{block\_host(h_3)\}\}$. Similarly, it is easy to show that the algorithm finally returns $\sigma(root(h_4)) = \{\{block\_host(h_3)\}\}$ as the recommended hardening strategy, which in this case coincides with the optimal solution.

Open issues. The dynamic version of the problem where we must take into account information about ongoing attacks remains an open issue. This will require the additional capability of detecting and tracking cyber attacks in real time as well as assessing and mitigating their potential impact on deployed missions. That is, given a mission, the set of hosts and links used to deploy the mission, and a stream of security alerts, we must find a cost-optimal time-varying strategy to harden, at any point in time, only the subset of resourced not used yet. A solution to this problem will help minimize the disruption that network hardening may cause to legitimate users.

5 Fault Tolerance Management

Fault tolerance is a critical and highly desirable property for mission deployed in the Cloud, given that large Cloud installations may be subject to a large number of failures. In this section, we adopt the perspective discussed in [12], where mission tasks can acquire desired fault tolerance properties as a service from a third-party (the fault tolerance *service provider*). The service provider must perform the following activities in order to realize this perspective.

- Defining an approach to implement general fault tolerance mechanisms as independent modules such that each module can transparently function on mission tasks.
- Analyzing the fault tolerance properties of each module by taking into account the failure behavior and system architecture. This sub-problem allows the service provider to select appropriate low-level modules based on the users' high-level goals.
- Defining a scheme to deliver a holistic fault tolerance solution to mission tasks by combining the set of selected modules.

Realizing fault tolerance modules. To offer fault tolerance as a service, the service provider must define general fault tolerance mechanisms in a way that they can transparently function on mission tasks deployed on virtual machines. This requirement can be satisfied by applying fault tolerance mechanisms at the virtualization layer [13]. We use ft_unit to denote the fundamental module that applies a coherent fault tolerance mechanism at the granularity of a VM instance. For instance, an ft_unit may replicate the entire VM instance on multiple physical hosts or an ft_unit may detect server crashes using well-known failure detection algorithms (e.g., by running the heartbeat protocol in the VM independently of mission tasks). In this manner, replication and failure detection can be performed without making any changes to the mission's source code, and the impact of hardware failures on the mission can be handled transparently.

Since different fault tolerance units realize different mechanisms, they offer a unique set of fault tolerance properties. Such properties can be characterized using their functional, operational, and structural attributes. The fault tolerance property p of an ft_unit can be denoted as $p = (u, \hat{p}, A)$ where u represents the ft_unit, $\hat{p}$ is the abstract property (e.g., availability, reliability), and A is a set of attributes that refers to the granularity at which u can handle failures, benefits and limitations of using u, and quality of service parameters. An order relationship can be defined on the domain of each attribute $a \in A$. Therefore, by looking at the attributes set A associated with an ft_unit, the service provider can evaluate fault tolerance properties that can be achieved with its use. An example of fault tolerance property for an ft_unit u^* is $p = (u^*, availability = 98\%, \{mechanism = \text{active\_replication}, no\_of\_replicas = 4, fault\_model = \text{node\_crashes}\})$.

Analyzing the effectiveness of a fault tolerance module. The effectiveness of an ft_unit can be evaluated in terms of the level of reliability and availability that can be obtained with its use. This analysis requires the service provider to (i) evaluate different configurations of modules, and (ii) quantify the reliability and availability obtained with each ft_unit by taking into account the failure characteristics of the system. We briefly discuss each of these two aspects in the following.

Evaluating the configuration of fault tolerance modules. A fault tolerance module ft_unit may have different configurations. For example, the ft_unit realizing replication schemes may have three configurations, namely, semi-active, semi-passive, and passive. These configurations represent the majority of fault tolerance implementations that are currently being used, and each configuration provides a different set of properties. One approach to characterize the effectiveness of an ft_unit, in a specific configuration, is to use Markov chains. As an example, we discuss Markov modeling for semi-active replication. Other models can be generated in a similar manner [8].

In semi-active replication, the input is either provided to all the replicas or state information of the primary replica is frequently sent to backup replicas. All the replicas (primary as well as backup replicas) execute all the instructions. However, only the output generated by the primary replica is made available to the user, and output messages from the backup replicas are logged. When the primary replica fails, one of the backup replicas can resume execution from a correct state.

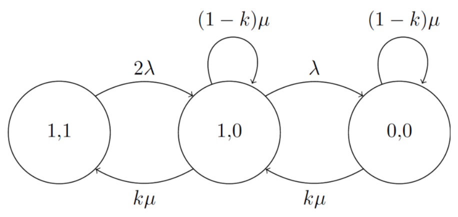

Fig. 3 Example of a Markov model for semi-active replication

Figure 3 illustrates the Markov model of an *ft_unit* that realizes a semi-active replication scheme with two replicas. Each state is represented as (x, y) where $x = 1$ implies that the primary replica is working and $x = 0$ implies that it failed. Similarly, y represents the state of the backup replica. Normal execution starts in state $(1, 1)$ and remains in this state as long as both replicas are available. When either the primary replica or the backup replica fails, the system moves to state $(0, 1)$ or $(1, 0)$ accordingly, and the other replica continues the execution. From the mission's perspective, states $(0, 1)$ and $(1, 0)$ are equivalent, thus, they are represented using a single state. In state $(0, 1)$ or $(1, 0)$, the recovery mechanism is initiated, and the system moves to state $(1, 1)$ if the recovery is successful. Otherwise, if the current replica fails, the system transitions to state $(0, 0)$ and the service becomes unavailable. In the figure, λ denotes the failure rate and μ denotes the recovery rate.

Deployment contexts for a fault tolerance module. If all the replicas generated through an *ft_unit* are deployed on the same physical host, the host failure may result in the failure of the mission. This implies that the location of each replica is also critical to the fault tolerance of the mission. We analyze how the fault tolerance property of a given *ft_unit* changes across three different deployment contexts.

- *Different physical hosts within a cluster*. Replicas of a mission task are assigned to different hosts that are connected in a LAN. This deployment provides benefits in terms of low latency and high bandwidth but offers very fault tolerance. For example, a single switch failure may prevent the replicas from communicating with one other, and as a consequence, the consistency protocol cannot be executed.
- *Different clusters within a data center*. Replicas of a mission task are assigned to hosts that belong to different clusters within the same data center. This deployment provides moderate benefits in terms of latency and bandwidth, and offers higher fault tolerance.
- *Multiple data centers*. Replicas of a mission task are assigned to hosts that belong to different data centers. This deployment reduces the performance of the mission with respect to network latency, but offers a very high level of fault tolerance.

As the values of most low-level parameters (e.g., MTBF, MTTR) of hardware and system software are normally vendor-confidential, the work presented in [8] uses the data published in [14, 15] to determine the overall *availability* provided by various ft_unit's for different configurations and deployment schemes. In particular, the results from Markov model analysis are combined with the notion of deployment levels using hierarchical fault trees for server failures. We observe from the results that the availability of missions is higher when replicas are placed in two different data centers. The value is slightly lower for the deployment level where replicas are placed in different clusters within a data center and still lower when replicas are placed inside the same cluster. The overall availability obtained by semi-active replication is slightly higher than semi-passive replication, whereas passive replication appears to be the worst.

This analysis allows a service provider to identify the placement conditions inherent to each ft_unit. Such conditions can be specified in the form of fault tolerance constraints, that are then taken into account while deploying mission tasks in the infrastructure (e.g., using the technique in Section 3). Examples of fault tolerance constraints are as follows [16].

- *Restriction.* The service provider may require that task replicas be located within a subset of hosts in the infrastructure (e.g., a cluster or data center). Such requirement naturally arises when a deployment context is chosen (e.g., place two replicas of a task in different clusters within a data center). To satisfy such requirements, the service provider can use a restriction constraint that limits a task $t_i \in T$ to being allocated only on a specified group of physical hosts $H \subset \mathcal{H}$. When the set $Restrict = \{(t_i, H_j) \mid t_i \in T \wedge H_j \subset \mathcal{H}\}$ is defined, the allocation function $a : T \to \mathcal{H}$ must ensure the following:

$$\left(\forall t_i \in T, H_j \in 2^{\mathcal{H}}\right) \;\; ((t_i, H_j) \in Restrict \implies a(t_i) \in H_j) \tag{11}$$

- *Forbid.* The service provider may need to specify that the allocation function must not deploy a given task on a subset of hosts. For example, if tasks t_1 and t_2 must be allocated on two different clusters C_1 and C_2, it is sufficient to restrict one task to one of the two clusters and forbid the other task from being deployed in the same cluster. Therefore, when the service provider defines a set $Forbid = \{(t_i, H_j) \mid t_i \in T \wedge H_j \subset \mathcal{H}\}$ specifying that task t_i must be forbidden from being allocated on hosts in H_j, the allocation function must satisfy the following:

$$\left(\forall t_i \in T, H_j \in 2^{\mathcal{H}}\right) \;\; (t_i, H_j) \in Forbid \implies a(t_i) \notin H_j) \tag{12}$$

- *Network latency threshold.* To balance the performance of the mission, the service provider may want to allocate task replicas $t_i, t_j \in T$ such that the network latency between them is below a given threshold δ. In this case, the service provider can define a set $Latency = \{(t_i, t_j, \delta) \mid t_i, t_j \in T) \wedge \delta \in \mathbb{R}^+\}$, and the allocation function $a : T \to \mathcal{H}$ must satisfy the following:

$$(\forall t_i, t_j \in T) \;\; ((t_i, t_j, \delta) \in Latency \implies latency(a(t_i), a(t_j)) \leq \delta) \quad (13)$$

We assume that the service provider realizes a range of fault tolerance mechanisms as ft_unit's and estimates the overall reliability and availability that can be achieved using each ft_unit with different configurations and deployment schemes. Let U be the set of possible ft_unit's applicable to the system. For a given user request, first, the set $U' \subseteq U$ of ft_unit's that satisfy the abstract property requirements is derived. Any $u \in U'$ can be used to deliver the desired fault tolerance properties if there are no additional constraints on cost or performance. However, since users may specify constraints on attributes in A, a second set $U'' \subseteq U'$ of modules is defined by only including those modules in U' that satisfy the additional constraints. For instance, a user may specify that the value of a given attribute $a \in A$ must be above a given threshold. Finally, modules in U'' are ordered with respect to users' requirements. The first ft_unit in the ordered set U'' can be selected as the most appropriate fault tolerance module.

Delivering comprehensive fault tolerance solutions. Although an ft_unit can serve as the fundamental fault tolerance module, a comprehensive solution ft_sol can be obtained by combining a set of ft_unit's in a specific execution logic. For example, a heartbeat test (ft_unit_1) can be applied only after a mission task is replicated on multiple hosts (ft_unit_2), and a recovery mechanism (ft_unit_3) can be applied only after a failure is detected. Therefore, using the above matching process, the service provider first designs a comprehensive fault tolerance solution ft_sol and applies it to the mission tasks. Note that by using ft_unit's to deliver a comprehensive solution, the extent of the fault tolerance support can be changed dynamically. In other words, the fault tolerance properties applied on a mission task can be dynamically changed based on the business needs. For instance, a robust failure detection mechanism can be replaced with a less robust one. Furthermore, by designing an ft_unit to be configurable at runtime, resource consumption and costs can be controlled.

The service provider starts monitoring the service once an ft_sol is applied to a mission. Runtime monitoring is critical for efficient service delivery since the context and attribute values of a fault tolerance solution may change at runtime due to the dynamic nature of the Cloud computing environment. To achieve this, the service provider first defines a set of rules over attributes $a \in A$ such that the validity of every rule establishes that property p is supported by the fault tolerance solution. For instance, given an ft_sol s_1 that satisfies property $p_1 = (s_1, availability = 98\%, \{mechanism =$ active_replication, $failure\_detection =$ heartbeat_test, $max\_recovery\_time =$ 25ms, $level = 3\})$, the set of rules that can test the validity of p_1 can be defined as:

- $r_1 : no\_of\_server\_instances \geq 3$

- $r_2 : heartbeat\_test\_frequency = 5ms$
- $r_3 : recovery\_time \leq 25ms$

In this context, the task of the service provider is to monitor the attribute values of each ft_sol at runtime, and verify the corresponding set of rules to ensure that missions requirements are satisfied.

6 Conclusions

In this chapter, we highlighted that existing solutions do not suitably address users' security and fault tolerance concerns in the Cloud computing scenario. We then showed how our work can address some of these limitations, although some issues still remain open.

Specifically, we formulated the mission deployment problem as a security-oriented task allocation problem, and proposed a solution aimed at minimizing a mission's exposure to vulnerabilities. In order to define a more comprehensive solution, and provide better availability and fault tolerance guarantees to missions, we discussed an efficient approach to effective network hardening. Finally, we discussed how to offer fault tolerance as a service to missions.

In addition to the open issues already discussed throughout the chapter, another important issue that needs to be addressed is the ability to automatically respond to incidents at runtime in order to salvage missions that may have already been compromised by those incidents.

References

1. P. Samarati and S. De Capitani di Vimercati, "Data protection in outsourcing scenarios: Issues and directions," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2010)*, Beijing, China, April 2010, pp. 1–14.
2. 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.
3. 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.
4. P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, May 2011.
5. 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.

6. K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, Indianapolis, IN, USA, 2010, pp. 93–204.
7. P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proceedings of the ACM SIGCOMM 2011*, Toronto, ON, Canada, August 2011, pp. 350–361.
8. R. Jhawar and V. Piuri, "Fault tolerance management in iaas clouds," in *Proceedings of the IEEE First AESS European Conference on Satellite Telecommunications (ESTEL 2012)*, Rome, Italy, October 2012.
9. D. S. Kim, F. Machida, and K. S. Trivedi, "Availability modeling and analysis of a virtualized system," in *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2009)*, Shanghai, China, November 2009, pp. 365–371.
10. M. Albanese, S. Jajodia, R. Jhawar, and V. Piuri, "Reliable mission deployment in vulnerable distributed systems," in *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2013)*, Budapest, Hungary, June 2013.
11. M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro, "A security analysis of amazon's elastic compute cloud service," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC 2012)*, 2012, pp. 1427–1434.
12. R. Jhawar, V. Piuri, and M. Santambrogio, "Fault tolerance management in cloud computing: A system-level perspective," *IEEE Systems Journal*, vol. 7, no. 2, pp. 288–297, June 2012.
13. B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2008)*. San Francisco, CA, USA: USENIX Association, 2008, pp. 161–174.
14. W. E. Smith, K. S. Trivedi, L. A. Tomek, and J. Ackaret, "Availability analysis of blade server systems," *IBM Systems Journal*, vol. 47, no. 4, pp. 621–640, 2008.
15. A. Undheim, A. Chilwan, and P. Heegaard, "Differentiated availability in cloud computing slas," in *Proceedings of the 12th IEEE/ACM International Conference on Grid Computing (GRID 2011)*, Lyon, France, September 2011, pp. 129–136.
16. 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.