

# A DESIGN METHODOLOGY FOR DEVELOPING RESILIENT CLOUD SERVICES

**C. Tunc<sup>1</sup>, S. Hariri<sup>1</sup>, and A. Battou<sup>2</sup>**

<sup>1</sup>University of Arizona, Tucson, AZ, United States <sup>2</sup>National Institute of Standards and Technology (NIST), Gaithersburg, MD, United States

## 9.1 Motivations

Cloud computing is a “model for enabling ubiquitous, convenient, on demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and delivered with minimal managerial effort or service provider interaction” [1]. The most widely accepted delivery models of Cloud Computing are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [2]. In addition, there are several deployment models (public, private, and hybrid) [3] and several emerging delivery models such as Storage as a Service (StaaS) [4], Security as a Service [5], and Network as a Service [6].

The recent embrace of cloud computing and services due to their performance and cost considerations will further exacerbate the security problem.

In cloud computing, organizations relinquish direct control of many security aspects to the service providers such as trust, privacy preservation, identity management, data and software isolation, and service availability. In addition, cloud computing integrates many technologies including virtualization, Web technologies, utility computing, and distributed data management, each with its own set of vulnerabilities. The adoption and proliferation of cloud computing and services will be severely impacted if cloud security is not adequately addressed. Traditional security approaches will not work effectively enough in a cloud environment due to many challenges related to the

monoculture paradigm [7] that is widely used in configuring cloud resources and services, the rapid and dynamic changes in cloud environments, the use of social networking software tools that can lead to rapid spread of viruses and worms, the manually intensive management of security policies, and the use of heterogeneous and mobile tools and devices [8].

Cloud security suffers from a wide range of attacks targeting from physical machines to virtualized environment [9]. The dependency of cloud computing on the virtualized environment raises more security issues, like hypervisor exploitations [10,11]. In addition, one of the main security issues in cloud computing is the insider attacks, which has increased with the exchange of cloud data between different organizations.

Some previous works have presented classifications of Cloud Security [12–14]. In IaaS, since infrastructure resources such as computation, storage, network, etc., are shared among multiple users, IaaS services may not be designed to provide strong isolation among tenants, malicious insiders can gain access of legitimate user's data [14]. For PaaS, the providers offer platforms for development and deployment of clients' own applications on the cloud and abusive use of APIs can threaten all service-models [14]. In SaaS, customers remotely connect to cloud to use provided software applications. Cross-site scripting [15], access control weaknesses, OS and SQL injection flaws, cross-site request forgery [16], etc., are the threats to the SaaS cloud model and data [17].

Since in cloud the customers' data reside on the third-parties' data-centers, data security is a major concern for clients and some researchers have addressed data security in their works [18–20]. In addition, the insider attacks still remain a high-risk threat from employees inside the cloud provider company which potentially have access to a huge source of customers' information, especially for mission critical systems. Also, DDoS (Distributed Denial of Service) or network attacks can threaten the availability of cloud services. For such cases, intrusion detection approaches such as [21,22] have been used.

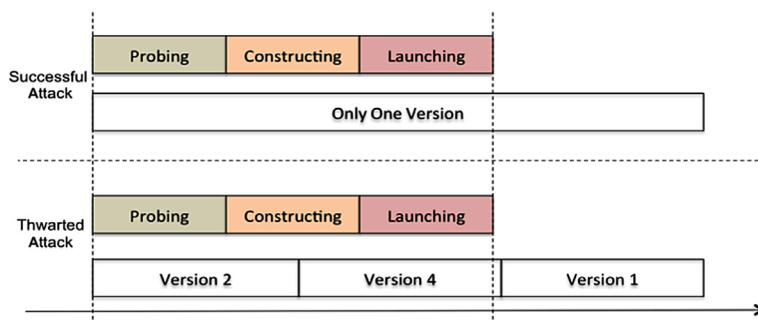
While various solutions have been proposed to solve cloud security issues [18,19], there is no comprehensive solution that covers all aspects of cloud security. Most of the offered solutions are partial and apply the detect-response model that fails with time. Furthermore, it is widely recognized that it is impossible to have cloud resources and services that cannot be penetrated and exploited. To address the cloud security challenges, we need an innovative design methodology based on resilience paradigm, moving target defense (MTD), and autonomic computing that will change the game to advantage the defender over the attacker.

## 9.2 Resilient Cloud Services Design Methodology

Our approach is based on using MTD concept to develop resilient cloud services (RCS) and algorithms that overcome the cloud security challenges [3]. The vision of MTD is defined as “Create, evaluate and deploy mechanisms and strategies that are diverse, continually shift, and change over time to increase complexity and costs for attackers, limit the exposure of vulnerabilities and opportunities for attack, and increase system resiliency” [23]. Our design methodology approach will make it much more difficult for any attacker to exploit vulnerability in a cloud service by changing the attack surface of the service randomly. Consequently, by the time an attacker probes, constructs, and launches an attack against the probed cloud service, that service will no longer exist or running; hence, the attack will become ineffective to disrupt the normal operations of the cloud service. Fig. 9.1 shows two scenarios: (1) *Successful Attack* when the cloud service environment stays static, as it is the case in most implementations/environments, giving the attacker plenty of time to study existing vulnerabilities by probing the services, and then constructing and launching an attack and (2) *Unsuccessful Attack* when the life span of one version of the cloud service is smaller than the time required to probe, construct, and launch an attack.

Our design methodology approach effectively utilizes the following capabilities:

1. *Redundancy*: It is a commonly used in fault tolerance technique [24] to continue to operate successfully in spite of software or hardware faults. In our approach, we combine the N-version programming [25] with hardware and virtual machine (VM) redundancy such that each cloud application



**Figure 9.1** Attack window for moving target defense.

task runs on different physical nodes as well as on different VMs in the cloud infrastructure.

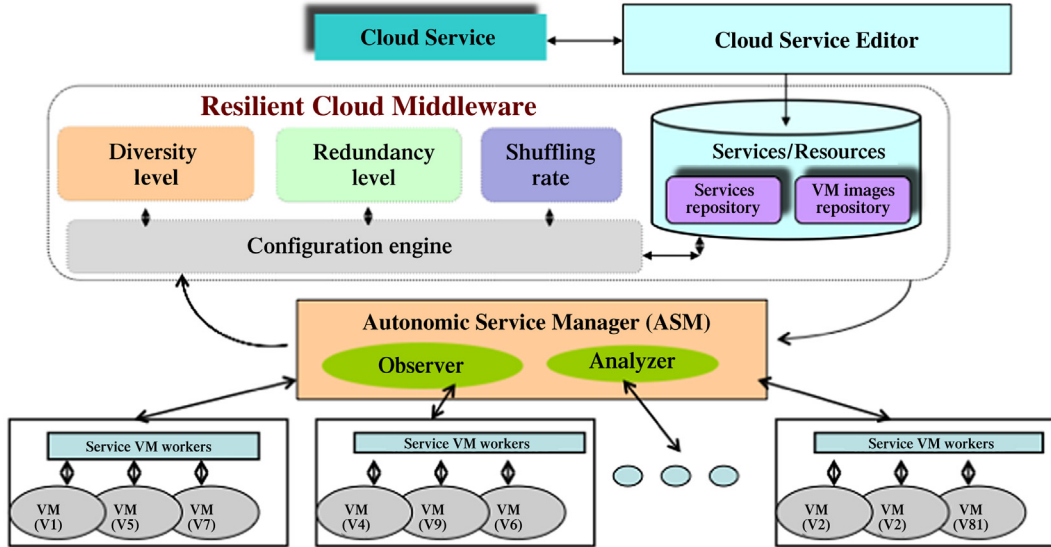
2. *Diversity*: This capability enables us to generate multiple functionally equivalent, behaviorally different software versions (e.g., each software task can have multiple versions, where each version can be a different algorithm implemented in a different programming language (e.g., C, Java, C++, etc.) that can run on different computing systems. We use the Compiler for Portable Checkpointing [26] to capture the current state of the cloud application such that it can be resumed on different cloud environments.
3. *Shuffling*: By randomly shuffling the use of diversified and redundant implementation of the cloud services, we obfuscate the execution environment of cloud services so that attackers will not be able to identify the type of execution environment and the resources used to run the cloud services. This approach will significantly reduce the ability of an attacker to disrupt the normal operations of a cloud service. Also, it allows adjusting the resilience level by dynamically increasing or decreasing the shuffling rate and their execution environments. A major advantage of this approach is that the dynamic change in the execution environment will hide the software flaws that would otherwise be exploited by a cyberattacker.
4. *Autonomic Management (AM)*: The primary task of the AM is to support dynamic decision among the various components such that the cloud resources and services. This way the services are dynamically configured to effectively exploit the current state of the cloud system and meet the application security requirements that might change at runtime.

## 9.3 RCS Architecture

Fig. 9.2 illustrates the architecture to implement the RCS development methodology using the following four main modules: Cloud Service Editor (CSE), Resilient Cloud Middleware (RCM), Configuration Engine (CE), Autonomic Service Manager (ASM) and Virtual Machines (VMs) to implement each resilient service managed by the ASM module.

### 9.3.1 Cloud Service Editor

The editor allows users and/or cloud service developers to specify the resiliency requirements of the cloud services by: (1) defining required diversity level (number of different versions



**Figure 9.2** Resilient cloud service architecture.

and/or different platforms); (2) defining redundancy level (how many redundant physical machines are required); and (3) defining how often the execution environment and phases need to be changed.

### 9.3.2 Resilient Cloud Middleware

The RCM provides the control and management services to deploy and configure the software and hardware resources required to achieve the resiliency specified by the CSE. The resilient operation for any cloud application is achieved by randomly shuffling the versions and the resources used to run each service so that it can hide (analogous to data encryption) the execution environment. The dynamic change in the service behavior makes it extremely difficult for an attacker to generate a profile with the possible flaws and to launch a successful attack. The decisions regarding when to shuffle the current variant, shuffling frequency, and variant selection for the next shuffle are guided by a continuous monitoring and analysis of current execution state of cloud services and the desired resiliency requirements.

To speed up the process of selecting the appropriate resilient algorithms and execution environments, the RCM repository contains a set of BO algorithms and images of VMs that run different operating systems (e.g., Windows, Linux, etc.) to

implement supported cloud services such as MapReduce, Web services, Request and Tracker applications, just to name a few.

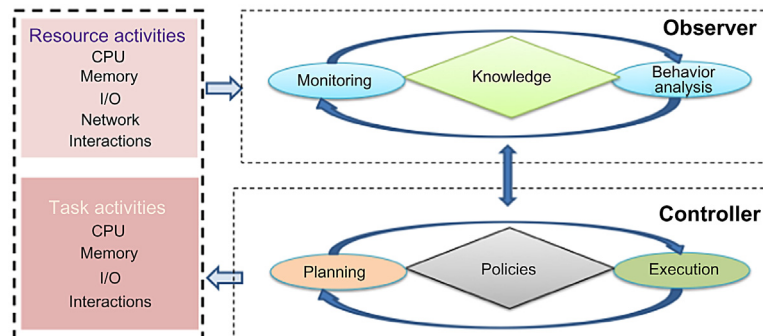
The CE takes the resiliency requirements specified by the users using the editor and uses the RCM repository to build the execution environment for RCS. The selected BO algorithm runs each Cloud Application or service as a sequence of execution phases, where each phase is administered by the Autonomic Service Manager (ASM). The ASM controls the operations of redundant and diversified VMs such that it can be resilient to any type of attack against the managed cloud services. Furthermore, we use Master Virtual Machines (MVMs) and Worker Virtual Machines (WVMs) where each MVM manages the voting algorithm on the results produced by several MVMs.

### 9.3.3 Autonomic Service Management

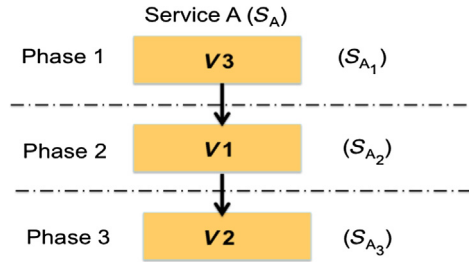
We have successfully designed and implemented a general autonomic computing environment (Autonomia) that will be leveraged in this task to implement the AM module [27]. By adopting the Autonomic architecture shown in Fig. 9.3, we implement the ASM using two software modules: Observer and Controller modules. The Observer module monitors and analyzes the current state of the managed cloud resources or services.

The Controller module is delegated to manage the cloud operations and enforce the resilient operational policies. The Observer and Controller pair provides a unified management interface to implement the required RCS.

In what follows, through an example, we show how we achieve the resilient operations by obfuscating the versions and the resources used to run each service. Let us assume we have a service A that we like to run in three phases as  $S_A = \{S_{A_1}, S_{A_2}, S_{A_3}\}$  as shown in Fig. 9.4.



**Figure 9.3** Autonomic service management architecture.



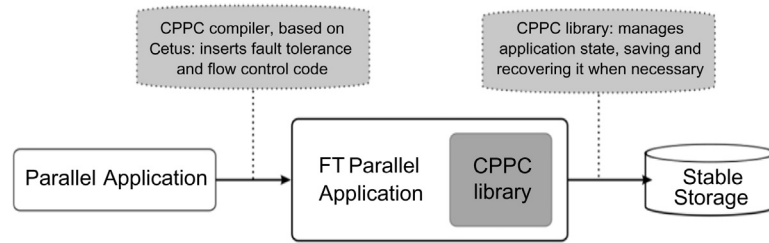
**Figure 9.4** Service Behavior Obfuscation example.

The Service Behavior Obfuscation (SBO) algorithm that will be managed by the ASM hides the execution environment by dynamically changing the sequence of execution of service versions by shuffling the service version running after each execution phase. The decisions regarding when to shuffle the current version, the shuffling frequency, and the version selection for the next shuffle are guided by a continuous feedback from the autonomic service manager. The service  $S_A$  runs in three diversified phases:  $V3$  in Phase 1,  $V1$  in Phase 1, and  $V2$  in Phase 3 of service  $S_A$ .

In addition to the shuffling of the execution of the service versions, we also apply hardware redundancy and software diversity to the implementation of the application tasks. The concept of design diversity is commonly used in software fault tolerance techniques to continue to operate successfully in spite of the software design faults. In our service obfuscation implementation approach, we combine N-version programming [25] and online anomaly behavior analysis techniques [28]. The multiversion implementation will prevent adversarial attacks from exploiting the monoculture problem. The anomaly behavior analysis approach will enable us to ensure that the operations of each task are completed correctly at the end of each execution phase; by using normal runtime models of the execution environment, we can detect any malicious changes in the execution environment, task variables, memory access range, etc.

To support the capability to resume the execution of different service versions on different platforms when they resume their execution, we use Compiler Portable Check-Pointing (CPPC) [26] technique in the service obfuscation algorithm. Checkpointing is widely used to recover from fault once it is detected as in fault-tolerance computing [29,30]. It periodically saves the computation state to a stable storage so that the application execution can be resumed by restoring such a state. The distinguishing characteristic of CPPC is that it allows for

**Figure 9.5** Compiler for portable checkpoint generation [26].



execution restart on different architectures and/or operating systems. It also attempts to optimize the amount of data saved to disk to improve efficiency and data transfers over the network. (CPPC is an open-source tool, available at <http://cppc.des.udc.es> under GPL license.) CPPC provides portable restart of applications in heterogeneous environments. Generated state files can be used to restart the computation on an architecture (or OS) different from the one that generated the file. The CPPC framework consists of a runtime library containing checkpoint-support routines, together with a compiler that automates the use of the library. The global process is presented in Fig. 9.5.

### 9.3.4 Resilience Analysis and Quantification

The process of quantifying the resilience that can be achieved by any SBO algorithm is a difficult process due to the heterogeneity of the environment, so a general and quantitative set of metrics for the resilience of cybersystems is impractical. In this section, we described an analytical approach to quantify the resilience that can be achieved using one configuration of the SBO algorithm. The method to quantify the resilience of a cloud application uses four important metrics: confidentiality, integrity, availability, and exposure.

The attack surface of a software system is an indicator of the system's vulnerability. So the higher the attack surface for a system, the lower the security is for that system [27]. The attack surface represents the area in which adversaries can exploit or attack the system through attack vectors. In our SBO-based resilient environment, the attack surface measurement can be used to quantify the resilience of the SBO algorithm with a given redundancy and redundancy level and number of phases. The goal of our analytical quantification approach is to show how the SBO algorithm can decrease the attack surface, and therefore, increase the resilience compared to a static execution environment. The first step in quantifying the attack surface is

identifying the software modules and libraries that can be exploited by attacks; this includes the operating systems, programming languages, and the network. There are many tools that can be used to identify attack vectors that exploit vulnerabilities in these software modules, such as Microsoft Attack Surface Analyzer [31], Flawfinder [32], Nessus [33], Retina [34], and CVEChecker [35]. In addition to these software systems and modules, the cloud application will also have an attack surface less than or equal to the system attack surface because the application while it is running will utilize a subset of the system attack surface; not all of the system attack vectors will be utilized by the application execution environment.

Common Vulnerabilities and Exposures (CVE) [36], which is a public reference for information security, vulnerability and exposures, is used to determine the confidentiality, integrity, and availability of the software system. Common Vulnerability Scoring System (CVSS) [37] is used as a standard measurement system for industries, organizations, and governments that need accurate and consistent vulnerability impact scores. Cyber resilience depends on maintainability, dependability, safety, reliability, performability, and survivability which are all functions of confidentiality, integrity, and availability [38].

In our approach to determine the attack surface, the following steps are used:

1. Scanning the cloud system using multiple attack vectors which are built based on known attack scenarios.
2. CVSS is used for identifying the characteristics and impacts of the system vulnerabilities using three groups: Base, Temporal, and Environmental scores. The Base group represents the intrinsic qualities of vulnerability. The Temporal group reflects the characteristics of a vulnerability that changes over time. The Environmental group represents the characteristics of vulnerability that are unique to any user's environment. Each group produces a numeric score (showing a range from 0 to 10) and a vector (compressed representation of the attack values used to derive the score). Using the CVSS database, it is possible to obtain the corresponding score by comparing the attack vectors.
3. Determine the attack vector impact and its probability.
4. Finally, determine the probability of an attacker to successfully exploit the identified vulnerabilities in the cloud system as well as in the cloud application.

In what follows, we describe in further detail our approach to evaluate analytically the probability of successful attacks against existing vulnerabilities when the SBO methodology is used.

In this analysis, we define the resilience as follows:

**Definition:** The system resilience  $R$  is defined as the ability of the system to continue providing its Quality of Service as long as the impact of the attacks is below a minimum threshold  $R$ .

The impact  $i_v(t)$  of a vulnerability  $v$  at an instant  $t$  is:

$$i_v(t) = \begin{cases} 0, & t < T_v \\ I_v, & t \geq T_v \end{cases}$$

where  $T_v$  is the time required for discovering the vulnerability and exploiting it, and  $I_v$  is the impact of exploiting the vulnerability.

The expected value of the impact of a vulnerability  $v$  is given by:

$$E[i_v] = I_v \cdot \Pr(A_v)$$

where  $A_v$  is the random variable that represents the occurrence of an attack exploiting vulnerability  $v$ . We can evaluate the probability of  $A_v$  as:

$$\Pr(A_v) = \Pr(A) \cdot \Pr(U_v)$$

$A$  denotes the existence of an attacker who is trying to exploit the system, and  $U_v$  denotes the time needed to successfully exploit the vulnerability  $v$ . To simplify the problem, we will assume that there will always be an attacker,  $\Pr(A) = 1$ , and any attacker that spends more than  $T_v$  time in exploiting vulnerability  $v$  is successful, that is, assume that all attackers are expert attackers and can successfully launch the attack in a minimum time  $T_v$ . By using the application life cycle time  $T_f$  and assuming that  $U_v$  is a uniform random variable, the pdf (probability density function) for  $U_v$  is given by:

$$\Pr(U_v) = \begin{cases} 0, & t < T_v \quad \text{or} \quad t > T_f \\ \frac{1}{T_f - T_v}, & t \geq T_v \end{cases}$$

We define the impact of a system with  $N$  vulnerabilities to be:

$$i_{\text{system}} = E[i_{v_1} + i_{v_2} + \dots + i_{v_N}]$$

Using the linearity property of the expected value, the previous equation can be re-written as:

$$\begin{aligned} i_{\text{system}} &= E[i_{v_1}] + E[i_{v_2}] + \dots + E[i_{v_N}] \\ &= I_{v_1} \cdot \Pr(A) \cdot \Pr(U_{v_1}) + I_{v_2} \cdot \Pr(A) \cdot \Pr(U_{v_2}) \\ &\quad + \dots + I_{v_N} \cdot \Pr(A) \cdot \Pr(U_{v_N}) \\ &= \sum_{k=1}^N I_{v_k} \cdot \Pr(A) \cdot \Pr(U_{v_k}) = \Pr(A) \cdot \sum_{k=1}^N I_{v_k} \cdot \Pr(U_{v_k}) \end{aligned}$$

Since, we do not have a direct control over the  $\Pr(A)$  or the impact value  $I_{v_k}$  of the  $k$ th vulnerability  $v_k$ , in our SBO technique, we continuously shuffle the application execution into multiple phases where we are basically reducing the execution time for each phase to be less than  $T_v$  for all or most vulnerabilities, which in turn forces  $\Pr(U_v)$  for those vulnerabilities to be zero. We are currently using the CVEChecker tool to get the impact score  $I_{v_k}$ .

Using the multiple functionally equivalent variants to run a cloud application will significantly improve its resiliency to attacks because that will reduce the probability of a successful attack on the application execution environment. For example, by using  $L$  functionally equivalent versions of the application, the probability of successfully exploiting an existing vulnerability  $v_k$  is given by:

$$\Pr(U_{v_k}) = \Pr(U_{v_k,1} \cap U_{v_k,2} \cap \dots \cap U_{v_k,L})$$

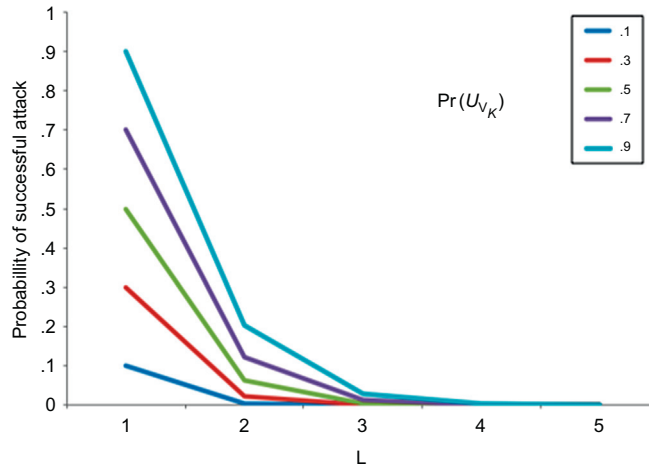
Since these versions are independent from one another:

$$\Pr(U_{v_k}) = \Pr(U_{v_k,1}) \cdot \Pr(U_{v_k,2}) \cdot \dots \cdot \Pr(U_{v_k,L})$$

Assuming that all versions are equally likely to be attacked:

$$\Pr(U_{v_k}) = \frac{1}{L} \Pr(U_{v_k}) \cdot \frac{1}{L} \Pr(U_{v_k}) \cdot \dots \cdot \frac{1}{L} \Pr(U_{v_k}) = \left( \frac{1}{L} \Pr(U_{v_k}) \right)^L$$

Fig. 9.6 shows the decrease in the probability of a successful attack as a function of the number of versions to be used in the SBO algorithm. For example, if we assume that the probability of an attacker to exploit a vulnerability equal to .5, by using two versions, this probability is reduced to .05 and by using three



**Figure 9.6** Probability of successful attack with respect to the number of versions.

versions, this probability is reduced to almost zero. Please note that the assumption if an attacker can exceed in exploiting existing vulnerability is very low using our SBO resilience approach, and consequently, a more a reasonable assumption is .1, and with that, we can see the probability of a successful attack against our approach drops to zero when we use only two versions.

From the previous discussion, it is clear that our technique will significantly reduce the ability of attackers to exploit existing vulnerabilities in cloud applications.

## 9.4 Experimental Results and Evaluation

### 9.4.1 Experimental Testbed Setup

Our testbed consists of IBM HS22 Bladecenter as our private cloud [39]. We have used three physical nodes on the Bladecenter for each of the applications that follow. On each of the three physical nodes, we have setup three VMs designated as follows:

- MVM: running Linux
- Slave 1 VM: running Linux
- Slave 2 VM: running Windows operating system

In addition, applications 2 and 3 consist of four additional VMs as follows:

- SBO Controller
- Supervisor (3 VMs are allocated where only one of them is active at any given time)
- Slave VMs: The two slaves contain functionally, equivalent, but behaviorally different versions of the same application.

For example, in *Figure*, each slave contains a C++ and a Java version of the MapReduce Application.

### 9.4.2 Applications Tested

#### 9.4.2.1 MapReduce

MapReduce [40] is widely used as a powerful parallel data processing model to solve a wide range of large-scale computing problems. With the MapReduce programming model, programmers need to specify two functions: Map and Reduce. The Map function receives a key/value pair as input and generates intermediate key/value pairs to be further processed. The Reduce function merges all the intermediate key/value pairs associated with the same (intermediate) key and then generates the final

output. There are three main roles: the master, the mappers, and the reducers. The single master acts as the coordinator responsible for task scheduling, job management, etc. MapReduce is built upon a distributed file system (DFS), which provides distributed storage. The input data is split into a set of map (M) blocks, which will be read by M mappers through DFS I/O. Each mapper will process the data by parsing the key/value pair and then generate the intermediate result that is stored in its local file system. The intermediate result will be sorted by the keys so that all pairs with the same key will be grouped together. The locations of the intermediate results will be sent to the master who notifies the reducers to prepare to receive the intermediate results as their input. Reducers then use Remote Procedure Call (RPC) to read data from mappers. The user-defined reduce function is then applied to the sorted data; basically, key pairs with the same key will be reduced depending on the user defined reduce function. Finally the output will be written to DFS.

Hadoop [41] is an open source implementation of the MapReduce framework and is used in our experimental results to evaluate our system for the MapReduce application. Oracle Virtualbox [42] has been used as the virtualization software. To maintain consistency with the MapReduce parlance defined in Ref. [40], we will refer to each physical host machine as master and each guest machine as slave. To prevent any single point of failure, each guest machine is configured to run in a single node cluster [41]. The MapReduce Wordcount program [40] is available on each slave in C++ and Java. Thus, the combination of *<physical machine, operating system, programming language>* represents a single version. Fig. 9.7 provides details about the application diverse versions used in our implementation.

The MapReduce application in our experiment is divided into three phases as follows:

- Phase 1: First Map function
- Phase 2: Second Map function
- Phase 3: Final MapReduce function.

The outputs of Phases 1 and 2 are used as inputs to Phase 3. During runtime, the application execution is performed in parallel on each of the three machines. Also, at the beginning of each phase, each master runs a local shuffler program to determine the version to run at the current phase. For this experiment, we have used a random number generator to determine the version that will run on each machine. At the end of each phase, the three masters run local acceptance tests. If the acceptance test fails, the output is taken from one of the other masters.

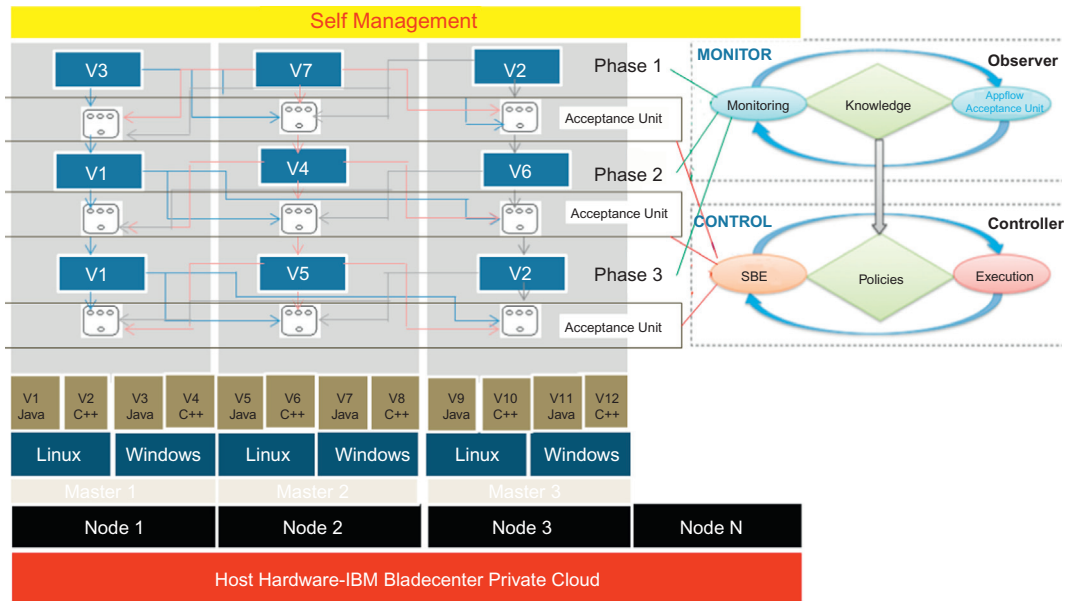
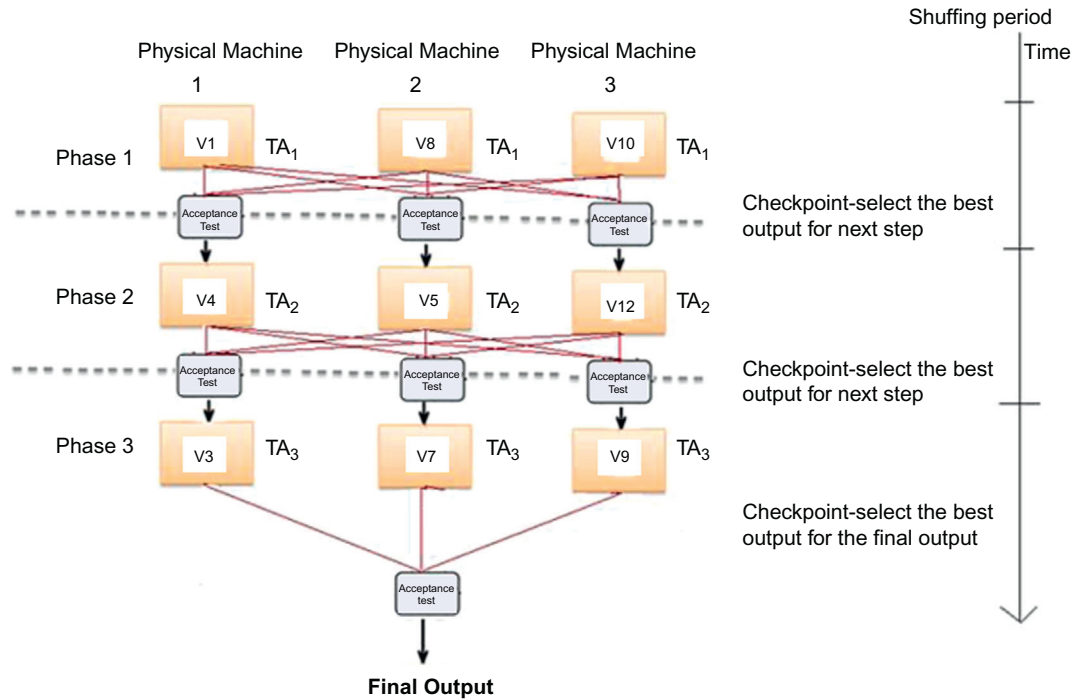


Figure 9.7 Resilient MapReduce application.

Fig. 9.8 shows an example of how to run the MapReduce application in a resilient manner using our methodology. At the beginning of Phase 1, the ASM runs a random number generator and selects versions V1, V8, and V10, respectively. After completion of the first Map on each physical machine, the output is checked for correctness by the acceptance test criteria. If this test fails, the ASM selects the output of Phase 1 from other physical machines and the first result that passes the acceptance test will be selected for the next phase of the application execution. Similar actions are performed in Phases 2 and 3.

#### 9.4.2.1.1 Case 1: Resilience Against Denial of Service Attacks

In this scenario, we launched a DoS attack on one of the machines used to run the MapReduce application. The ASM detects the DoS attack and tolerates it. Although the DoS attack affected the attacked physical machine and increased its response time by 23%, since we took the output from the other physical machine the response time of the application with and without attack remained the same. An overhead time of 14% of response time was added by our approach.



**Figure 9.8** Example of achieving resilient MapReduce application.

#### 9.4.2.1.2 Case 2: Resilience Against Insider Attacks

In this case, one of the machines (the fastest physical machine) is compromised by an insider attack and the computations running on that machine were changed by the internal attacker. Similar to the previous case, the application continued to operate normally in spite of the insider attack because the results from the compromised machine were ignored and the results from other versions were used instead. The performance impacts and overhead on the application performance are shown in [Table 9.1](#).

As shown in [Table 9.1](#), the average response time using the RCS approach increases by 14% (without attack) and 24% (with attack).

#### 9.4.2.2 Jacobi's Iterative Linear Equation Solver

Linear equations are used to solve a wide range of real world scientific and engineering problems. The Jacobi technique is an

**Table 9.1 MapReduce Result Summary**

	Response Time	CPU Utilization per Physical Machine	Memory Utilization per Physical Machine	Network Utilization (%)
Without RCS	A	B	C	0
With RCS and no attack	1.14A	1.08B	1.02C	1
With RCS and attack	1.24A	1.12B	1.04C	2

Operating System/ Programming Language	Windows		Linux		Linux	
	V1	V4	V7	V10	V13	V16
C	V1	V4	V7	V10	V13	V16
C++	V2	V5	V8	V11	V14	V17
Fortran	V3	V6	V9	V12	V15	V18

**Figure 9.9** Versions used in Application 2.

iterative technique for solving a set of linear equations under two assumptions [43]:

- The system given by  $Ax = B$  has a unique solution
- The coefficient matrix  $A$  has no zeroes on its diagonal.

To solve a set of  $n$  equations, we solve the first equation for  $x_1$ , second equation for  $x_2$  as follows: We first make an initial assumption of the values of  $x$ . We then substitute these values into the right-hand side of the above set of equations. This completes the first iteration. This process is repeated until convergence is reached on the values of  $x$ . The implementation runs on a three node cluster each hosting two VMs. One of these VMs is Windows based, while the other is Linux based. We have used VMware vSphere 5 [44] for the virtualization. The Jacobi Algorithm described above has been implemented in C, C++, and Fortran, thus creating multiple versions (see Fig. 9.9).

Table 9.2 summarizes the overhead in terms of the execution time and overhead percentage for five programs with a normal execution time ranging from 200 to 3600 seconds, respectively. The overhead is given as a function of the number of phases selected to run the application.

We calculated the overhead as the additional time taken with our algorithm compared to running the application without RCS. As shown in Table 9.2, for programs with higher execution

**Table 9.2 Overhead in Application 2**

Execution Time Without RCS(s)	Execution Time With RCS(s)					
	Two phases		Three phases		Four phases	
	Time	OH (%)	Time	OH (%)	Time	OH (%)
200	218	9	248	24	276	38
800	838	5	890	11	988	24
1500	1568	5	1624	8	1663	11
3600	3671	2	3847	7	3890	8

	Physical Machine Number					
	1		2		3	
Operating System	Linux	Windows	Linux	Windows	Linux	Windows
Version Number	V1	V2	V3	V4	V5	V6

**Figure 9.10** Versions used with the MiBench suite.

times, the overhead due to RCS reduces significantly. For example, for a program with execution time of 3600 seconds, the overhead percentage for three phases is 7%. The number of phases to run each application can be chosen such that it meets the performance and resilient requirements of the application.

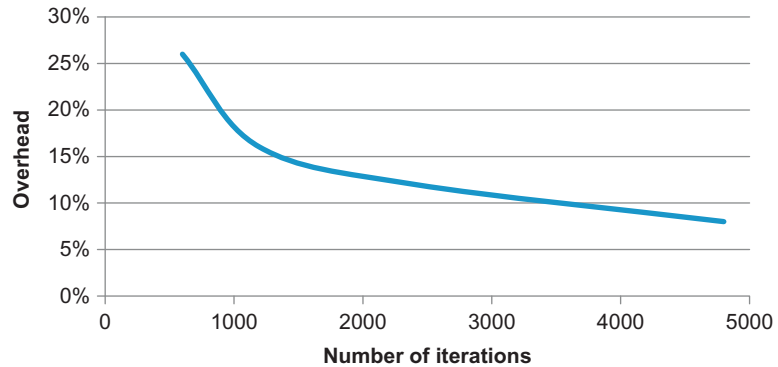
#### 9.4.2.3 MiBench Benchmarks

The MiBench Benchmarks [45] consist of C programs from six categories each targeting a specific area of the embedded market. We used the following applications from the MiBench benchmark suite:

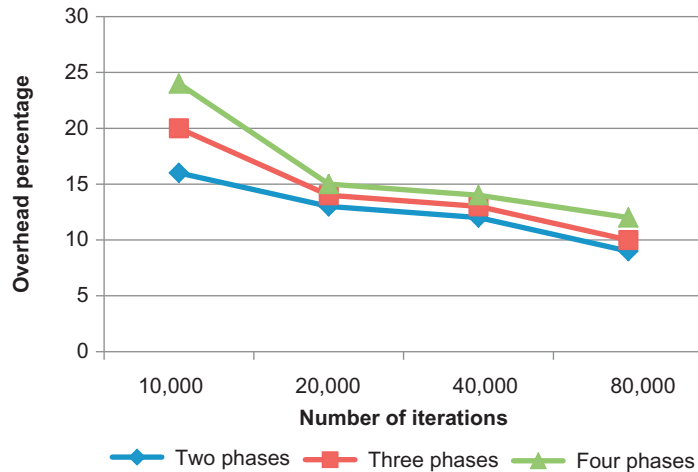
1. *Basicmath (Automotive and Industrial category)*: This program performs mathematical calculations like cubic function solving, integer square root, and angle conversions from degrees to radians are all necessary calculations for calculating road speed or other vector values.
2. *Dijkstra's algorithm (Network category)*: This program constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm.

For each of the above available C programs, we used diversity in operating systems to have a total of six versions. The versions used are shown in Fig. 9.10. We calculated the overhead of our RCS

**Figure 9.11** Basicmath—Overhead for RCS with three phases.



**Figure 9.12** Dijkstra's Algorithm—Overhead for RCS with three phases.



approach for different number of iterations of the abovementioned benchmarks. The results are presented in [Figs. 9.11 and 9.12](#). As seen in these figures, the overhead of our algorithm decreases as program size increases ([Figs. 9.11 and 9.12](#)).

## 9.5 Conclusions and Future Work

While cloud computing is emerging as a promising paradigm, security is a significant barrier to its adoption. In this chapter, we first presented an overview of the current security issues in cloud computing. We summarized previous works that classified cloud security issues on the basis of cloud delivery models and the components of the cloud. Further, we also observed that attacks on cloud systems cannot be prevented. We have presented a design methodology to develop RCS that based on the

following capabilities: Redundancy, Diversity, Shuffling, and Autonomic Management. In the RCS methodology, we adopt diversity technique to the cloud execution environment, redundancy in the resources used to run the cloud services and randomly changing the versions and resources used to make it prohibitively expensive for attackers to figure the current cloud service execution environment and succeeding in exploiting vulnerabilities and launching attacks. We also presented a testbed to validate the RCS architecture and resilient algorithms using three applications (MapReduce, Jacobii's iterative linear equation solver, and some programs from the MiBench benchmark suite). Our experimental results showed that our RCS approach can tolerate a wide range of attack scenarios with around 7% of overhead time. As a future research direction, we are currently working on developing analytics techniques to quantify the resilience of different RCS implementation strategies, overhead, and performance of the cloud services.

## Acknowledgments

This work is partly supported by the Air Force Office of Scientific Research (AFOSR) Dynamic Data-Driven Application Systems (DDDAS) award number FA95550-12-1-0241, National Science Foundation research projects NSF IIP-0758579, SES-1314631, and DUE-1303362, and Thomson Reuters in the framework of the Partner University Fund (PUF) project (PUF is a program of the French Embassy in the United States and the FACE Foundation and is supported by American donors and the French government).

## References

- [1] P. Mell, T. Grance, The NIST Definition of Cloud Computing, 2011.
- [2] L. Savu. Cloud computing: deployment models, delivery models, risks and research challenges, in: International Conference on Computer and Management (CAMAN), Wuhan, China, 2011.
- [3] S. Subashini, V. Kavitha, [A survey on security issues in service delivery models of cloud computing](#), *J Netw Comput Appl* 34 (2011) 1–11.
- [4] J. Wu, et al. Cloud storage as the infrastructure of cloud computing, in: Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on, IEEE, 2010.
- [5] C.P. Ram, G. Sreenivaasan, Security as a Service (SasS): securing user data by coprocessor and distributing the data, in: Trendz in Information Sciences & Computing (TISC), Chennai, 2010.

- [6] P. Costa, et al. NaaS: network-as-a-service in the cloud, in: 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, San Jose, CA, 2012.
- [7] K.P. Birman, F.B. Schneider, The monoculture risk put into context, in: Security and Privacy, IEEE, January–February 2009, pp. 14–17.
- [8] S. Hariri, M. Eltoweissy, Y. Al-Nashif, Biorac: biologically inspired resilient autonomic cloud, in: Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research, ACM, 2011.
- [9] Cloud Security Alliance. Security as a Service [online]. Available from: <<https://cloudsecurityalliance.org/research/secaas/>> (accessed January 2013).
- [10] M. Schmidt, L. Baumgartner, P. Graubner, D. Bock, B. Freisleben, Malware detection and kernel rootkit prevention in cloud computing environments, in: 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2011.
- [11] D. Goodin, Webhost Hack Wipes Out Data for 100,000 Sites [Online]. Available from: <[http://www.theregister.co.uk/2009/06/08/webhost\\_attack/](http://www.theregister.co.uk/2009/06/08/webhost_attack/)> (accessed January 2013).
- [12] V.S. Subashini, A survey on security issues in service delivery models of cloud computing, *J Netw Comput Appl* 34 (2011) 1–11.
- [13] R. Bhadauria, S. Sanyal, Survey on security issues in cloud computing and associated mitigation techniques, *Int J Comput Appl* 47 (18) (2012) 47–66.
- [14] C. Modi, D. Patel, B. Borisaniya, A. Patel, M. Rajarajan, A survey on security issues and solutions at different layers of Cloud computing, *J Supercomput* (2012) 1–32.
- [15] H. Zeng, Research on developing an attack and defense lab environment for cross site scripting education in higher vocational colleges, in: Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on, 21–23 June 2013, pp. 1971–1974.
- [16] M.S. Siddiqui, D. Verma, Cross site request forgery: a common web application weakness, in: Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on, 27–29 May 2011, pp.538–543.
- [17] G. Pék, L. Buttyán, B. Bencsáth, A survey of security issues in hardware virtualization, *ACM Comput. Surv.* 45 (3) (2013), Article 40 (July 2013), 34 pages.
- [18] M. Abbasy, B. Shanmugam, Enabling data hiding for resource sharing in cloud computing environments based on DNA sequences, in: IEEE World Congress, 2011.
- [19] J. Feng, Y. Chen, D. Summerville, W. Ku, Z. Su, Enhancing cloud storage security against roll-back attacks with a new fair multi-party non-repudiation protocol, in: Consumer Communications and Networking Conference, 2011.
- [20] L. Kaufman, Data security in the world of cloud computing, *IEEE Secur Priv* 7 (4) (2009) 61–64.
- [21] Y.B. Al-Nashif, A. Kumar, S. Hariri, Y. Luo, F. Szidarovszky, G. Qu, Multi-level intrusion detection system (ML-IDS), in: ICAC 2008, pp. 131–140, 2008.
- [22] P. Satam, H. Alipour, Y. Al-Nashif, S. Hariri, Anomaly Behavior Analysis of DNS Protocol, *Journal of Internet Services and Information Security (JISIS)* 5 (no. 4) (2015) 85–97.
- [23] [www.nitrd.gov](http://www.nitrd.gov), [online] May 13, 2010. <[http://www.nitrd.gov/pubs/CSIA\\_IWG\\_%20Cybersecurity\\_%20GameChange\\_RD\\_%20Recommendations\\_20100513.pdf](http://www.nitrd.gov/pubs/CSIA_IWG_%20Cybersecurity_%20GameChange_RD_%20Recommendations_20100513.pdf)> (cited: 15.01.13).
- [24] B. Randell, System structure for software fault tolerance, *IEEE Trans Softw Eng* 1 (1975) 220–232.

- [25] A. Avizienis, The N-version approach to fault tolerant software, *IEEE Trans Softw Eng SE-11* (12) (1985).
- [26] G. Rodríguez, M.J. Martín, P. González, J. Touriño, R. Doallo, CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications, *Concurr Comput* 22 (6) (April 2010) 749–766.
- [27] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, S. Rao Autonomia: an autonomic computing environment, in: Performance, Computing, and Communications Conference, Proceedings of the 2003 IEEE International, IEEE, 2003, pp. 61–68.
- [28] S. Hariri, G. Qu, *Anomaly-based self-protection against network attacks, Autonomic Computing: Concepts, Infrastructure, and Applications*. s.l, CRC Press, Boca Raton, FL, 2007, pp. 493–521.
- [29] A. Tyrrell, Recovery blocks and algorithm based fault tolerance, in: 22nd EUROMICRO Conference, 1996.
- [30] K.H. Kim, H.O. Welch, Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications, *IEEE Trans Comput* 38 (1989) 626–636.
- [31] [Online]. Available from: <<http://www.microsoft.com/en-us/download/details.aspx?id=24487>>.
- [32] [Online]. Available from: <<http://www.dwheeler.com/flawfinder/>>.
- [33] [Online]. Available from: <<http://www.tenable.com/products/nessus>>.
- [34] [Online]. <<http://www.beyondtrust.com/Products/RetinaCSThreatManagementConsole/>>.
- [35] [Online]. Available from: <<http://cvechecker.sourceforge.net>>.
- [36] [Online]. Available from: <<https://cve.mitre.org/>>.
- [37] [Online]. Available from: <<http://www.first.org/cvss/cvss-guide>>.
- [38] [Online]. Available from: <<http://search.cpan.org/~nwclark/perl-5.8.9/utills/perlcc.PL>>.
- [39] [Online]. <<http://www-03.ibm.com/systems/bladecenter/index.html>>.
- [40] D. Jeffrey, G. Sanjay, MapReduce: simplified data processing on large clusters, in: Sixth Symposium on Operating Systems Design and Implementation, 2008.
- [41] Apache Hadoop [Online]. <<http://hadoop.apache.org/>>.
- [42] Oracle VirtualBox [Online]. <<http://www.oracle.com/technetwork/server-storage/virtualbox/overview/index.html>>.
- [43] [Online]. <[http://college.cengage.com/mathematics/larson/elementary\\_linear/5e/students/ch08-10/chap\\_10\\_2.pdf](http://college.cengage.com/mathematics/larson/elementary_linear/5e/students/ch08-10/chap_10_2.pdf)>.
- [44] [Online]. <<http://www.vmware.com/products/vsphere/mid-size-and-enterprise-business/overview.html>>.
- [45] M.R. Guthaus, et al. Mibench: a free, commercially representative embedded benchmark suite, in: Proceedings of the Workload Characterization, Washington DC, 2001.