Self-Protection against Attacks in an Autonomic Computing Environment

Guangzhi Qu∗, Osamah A. Rawashdeh†, Dunren Che‡
Oakland University
Rochester, MI 48309
Southern Illinois University
Carbondale, IL 62901

Abstract
Computing systems have steadily evolved into more complex, interconnected, heterogeneous entities. Ad-hoc techniques are most often used in designing them. Furthermore, researchers and designers from both academia and industry have focused on vertical approaches to emphasizing the advantages of one specific feature such as fault tolerance, security or performance. Such approaches led to very specialized computing systems and applications. Autonomic systems, as an alternative approach, can control and manage themselves automatically with minimal intervention by users or system administrators. This paper presents an autonomic framework in developing and implementing autonomic computing services and applications. Firstly, it shows how to apply this framework to autonomically manage the security of networks. Then an approach is presented to develop autonomic components from existing legacy components such as software modules/applications or hardware resources (router, processor, server, etc.). Experimental evaluation of the prototype shows that the system can be programmed dynamically to enable the components to operate autonomously.

1 Introduction
The proliferation of networked systems and services coupled with the exponential increase of their complexity has complicated their control and management. In fact, current management technologies and software tools are becoming less efficient in dealing with such increased complexity, heterogeneity and dynamism, rendering systems and networks unmanageable and insecure. This has led researchers to consider alternative programming paradigms and management techniques based on strategies used in biological systems (e.g., autonomic nervous systems).

Next generation systems and applications need to be of high-performance, reliable, secure and cost-efficient. Ad-hoc integration of techniques attributed to improving performance, fault-tolerance, or security is usually costly and may not be optimized to highlight all the expected features. Actions designed to enforce security may diminish the impact of actions designed to improve performance. Further more, since few attributes may change dynamically during runtime (e.g., performance), the outcome of such integration may be unpredictable. Hence, it is imperative for future systems and applications to be adaptive and holistic in addressing all related attributes such as performance, security and fault tolerance.

Autonomic computing provides alternative design paradigms and management techniques to deal with complexity, dynamism, heterogeneity and uncertainty. It aims at realizing computing systems and applications that are capable of managing themselves with minimal human intervention. There have been several efforts [10, 12] to characterize the main features that make a computing system or an application autonomic. In general, an autonomic system must at least support the following four features: self-protecting, self-optimizing, self-healing, and self-configuring. In this autonomic computing paradigm, a component (e.g., hardware resource or a software module) can be used to build into a larger autonomic system where each (compound) component can be configured dynamically to support the required combination of the four properties.

The organization of the remaining sections of the paper is as follows. Section 3 presents the autonomic control and management framework and describes the detailed implementation of the autonomic components. In Section 4, evaluation of the performance and effectiveness is carried on a prototype system. Section 5 summarizes the paper and discusses future research activities.

2 Related Works
There have been a number of research efforts in designing and implementing autonomic computing systems and services. Based on the type of autonomic properties supported by the system and its adaptation approach, these efforts can be broadly classified into the following three categories: 1) Autonomic Middleware Techniques; 2) Autonomic Programming Paradigms; 3) Adaptation Approach.
2.1 Autonomic Middleware Techniques

In this approach, middleware and common services are added to existing legacy systems to support autonomic behaviors. OceanStore [22] is a utility infrastructure designed to provide continuous access to persistent information by incorporating policy based caching, routing substrate adaptation, autonomic replication, continuous monitoring, testing, and repairing. A number of IBM projects focused on incorporating self-management capabilities into their product modules such as SMART DB2 [17], which builds on DB2 and extends its existing self-management techniques. The Oceano [11] project involves designing and developing a pilot prototype of a scalable, manageable infrastructure for large-scale computing utility power plant that enables multi-
customer hosting on a virtualized collection of hardware re-

sources. Microsoft’s AutoAdmin [2] project aims at making database systems self-tuning and self-administering by tracking the usage of the underlying systems and gracefully adapting to meet applications’ requirements. IFLOW [15, 14] is an autonomic middleware for implementing distributed information flow services. It provides the functionality of dynam-
ically associating its abstract information graph to physical network nodes based on utility functions. VIOLIN [23] is a system that supports self-adaptation of allocation of comput-
ing resources across multi-domains by considering the dy-

namic availability of infrastructure resources and the cur-
rent applications’ resource requirement. Recovery Oriented Computing (ROC) [18] focuses on speeding up the recovery from hardware faults, software bugs, and operator errors. It reduces recovery time through a set of techniques such as automatic diagnosis, fine-grained partitioning and re-
cursive restart-ability. IBM Unity [5, 27] project proposes utility-function-based self-management. Astrolable [21] is a distributed information management system that permits an application to locate a resource, and offers a scalable way to track the states of the system as they evolve over time.

2.2 Autonomic Programming Paradigms

In this approach, users are provided with tools to compose components and follow adaptable programming paradigms to develop large autonomic applications that can dynamically change their structure and the algorithms used to implement their components in order to achieve the desired autonomic computing behavior. Liu et al. [16] describe the Accord Programming System that extends existing programming models to enable the development of self-managing grid applications by allowing the behavior of applications and sys-
tems to be dynamically specified at runtime. Heinis et al. [9] describe a self-configuring composition engine for grid and web services that achieves self-configuring, self-tuning and self-healing behaviors in the presence of varying workloads. Schwan et al. [24] describe the AutoFlow project designed to meet the critical performance requirements of distributed information flow applications.

2.3 Adaptation Approach

Autonomic systems can be classified based on the on the techniques used to implement the required system adapta-
tion into three types: Policy rule, Optimization, and AI plan-
ing and learning techniques. In the policy rule approach, condition-action policy dictates the action(s) that should be taken whenever the system is in a given state [1, 12, 13]. In the optimization approach, analytical techniques are used to model the overall system behavior and services through a utility function that is used to select the optimal adaptation stra-

tegy [27, 26, 14]. Rather than using the utility function as an objective function, [3] proposed an analytical queuing network model for resource allocation in a data center and ap-
pied combinatorial search techniques to determine optimal allocation. One limitation of optimization-based techniques is their requirement to derive an analytical model that cap-
tures accurately the system operations. In order to solve this difficulty, AI planning and learning techniques are used to model system behavior by using data mining and statistical techniques [4, 20, 28, 25].

3 System Architecture

In this work, we have built an autonomic control and management framework. Under this framework, self-
configuration and self-protection are implemented. In the following, the architecture is briefly reviewed first followed by the detailed discussion on implementing self-configuration and self-protection services.

3.1 Self-Configuration Service

The approach to automating software modules or net-
work resources is based on augmenting them with two enti-
ties [4]: CMI and CRM, as shown in Figure 1. It extends traditional components (e.g., Corba and Java beans) with pro-
visions to support autonomic features (e.g., self-configuring, self-protecting, and self-healing) and thus behave autononi-
ously. This architecture can either be used to implement auton
omous components from scratch or add automating provisions to existing software modules or resources.

CMI consists of four ports: Configuration port, Moni-
tor port, Actuator port and Policy port. It is a passive module that stores all control and management policies that govern the operation of components and their interaction with the en-
vironment. However, CRM represents the active module that aims at enforcing the policies specified in CMI at runtime. Consequently, CRM continuously monitors and analyzes the execution of its component and interrupts its operation when the execution environment of the current component cannot meet the desired operational and functional requirements. For
example, the required system resources (e.g., CPU time and memory) are unavailable to the components, or a failure/fault occurs in the environment. The planning module will determine the appropriate corrective actions that will be performed by the execution module.

3.1.1 Component Management Interface (CMI)

CMI provides four ports to specify the control and management requirements associated with each software component and/or network resource, as briefly described below:

- **Configuration Port**: It defines the configuration attributes required to automate the deployment of the component/resource and how to setup its execution environment. A component configuration attributes include its name, resource requirement, configuration parameters, and dependency specifications.

- **Monitor Port**: The monitor port defines all the measurement attributes that must be monitored at runtime. The monitoring capacity is provided by the autonomic component. Either pull or push mode can be configured to operate during the deployment.

- **Actuator Port**: It defines all the actions (e.g., stop execution, task migration, packet filtering) that can be performed on the component or resource in order to force the component/resource to behave autonomically.

- **Policy Port**: It describes the policies that define the normal/acceptable component/resource operations. These policies are described using two types of rules: Behavior Rules and Interaction Rules. The behavior rules describe the rules that govern the normal operations as a stand-alone component (e.g., CPU utilization, memory usage, CPU-MEM interactions), while the interaction rules specifies the rules that describe how this component should interact with its environment when it is behaving normally (e.g., when a component is compromised and started attacking its environment, such behavior should be detected by the component interaction rules). Operation port rules can be expressed in the following format: IF Condition1 AND Condition2 AND Conditionn THEN Action1 AND Action2 ⋯ Actionn. Each rule is expressed with a conjunction of one or more conditions and actions. A condition is the logical combination of a component/resource state and/or measurement attributes or an external event triggered by other components or resources. An action can invoke the sensor/actuator functions specified in the control port or send out events to other components.

3.1.2 Component Runtime Manager (CRM)

CRM is the local control system that continuously monitors the component operations, analyzes the running state, plans the appropriate corrective actions if needed, and executes these actions to bring the component back to an acceptable normal state of operation. For example, if the component state deviation caused by malicious attacks, the planning module will determine the appropriate protective security actions to mitigate and prevent the current attacks. Similarly, when the component state deviates from normal behavior due to failures, the planning module will determine the appropriate actions to recover from the failures and continue normal operations. Consequently, CRM handles all the desired component attributes in a holistic way rather than handling each attribute with separate mechanisms.

3.2 Self-Protection Service

The primary goal of the self-protection service is to detect network attacks accurately and proactively prevent or minimize their impact on the network operations and other services. The main modules to implement the self-protection service include Online Monitoring, Feature Selection, Anomaly Analysis, and Protective Action (see Figure 2).

3.2.1 Online Monitoring Module

The main goal of online monitoring and analysis modules is to analyze monitored metrics to determine whether or not network resources and computers are operating normally. The online monitoring module collects three types of data: 1) Network traffic (number of connections, number of open ports, packet size, and transfer rate), 2) Computer workload (CPU, memory, and I/O) requirements, and 3) Application/process...
workload information. The network traffic is obtained using the Cisco NetFlow [7], which is now the de facto standard for network wide monitoring. NetFlow consists of a packet sampling algorithm, and a NetFlow agent that combines flow samples, interface counters, and the state of the forwarding/routing table entries associated with each sampled packet into a NetFlow datagram which is forwarded to a central NetFlow collector. The central NetFlow collector receives a continuous stream of NetFlow datagrams from across the entire network and analyzes them to form a sufficient, real-time view of layer 3 to layer 4 traffic flows across the entire network. For computer workload, operating system logs are used to obtain the required workload information about computers and their processes. The monitored information will be fed to the anomaly analysis module for further processing.

### 3.2.2 Feature Selection Module

Feature selection can be viewed as a data filtering process aimed to choose the minimal subset of features relevant to a particular decision (e.g., whether the network behaves normally). Feature filtering can help reduce data dimensionality and thus the overhead of the online data monitoring and analysis modules. Evaluation of an individual feature shows the relevance of the feature with the final decision.

In the filtering approach, mutual information and a new correlation measure are used to quantify the dependency among features [19]. If the information obtained from a set of features influence the output of a given decision variable (e.g., the increase in number of connection requests and number of DNS requests might indicate the existing of a probing attack), the correlation between these features is quantified by a Decision Dependent Correlation (DDC) [19]. Based on the decision dependent correlation metric, a near optimal set of features with respect to the decisions (e.g., decision variable to indicate normal or abnormal behavior in network resources or computers, acceptable system performance, etc.) is selected.

### 3.2.3 Anomaly Analysis Module

The online monitoring module continuously feeds streaming data into the anomaly analysis module. The streaming data in terms of different features is calculated based on different timing intervals. Hence, the streaming features with different granularity can discover the sophisticated attackers who take advantage of the window boundary to avoid detection. The anomaly analysis module has two main functions. The first one is the classification function in which the learning function builds a profile of the normal network behaviors that can be described using rule sets. The second function is the prediction function that is used during the run time by the analysis module that applies the profile rule sets to the observed network behavior to determine the occurrence of any anomalous behavior that could be triggered by network attacks.

Beyond the raw NetFlow features, more are derived. The new features include \( \text{diff}_\text{src}\_rate \) (the different service rate from a specific host), \( \text{same}_\text{src}\_rate \) (the same service rate from a specific host), just to name a few. Using the feature selection method in Section 3.2.2, a small set of features were determined for each type of network attacks. Rule learning algorithm (RIPPER [8]) is utilized to establish the baseline model for normal network behaviors. The construction of the baseline models is very efficient. For a training dataset of \( 300 \) hours to generate the baseline model that consists of around 120 rules. A sample of the rule used in the anomaly detection algorithm is shown below.

![Algorithm 1: Sample Rules for Anomaly Detection](image)

### 3.2.4 Protective Action Module

The action module will take appropriate actions according to the analysis results through both Actuator port and Policy Port of CMI. The Policy port will produce recommendations (e.g., shut down some router interfaces to prevent communications with outside and thus the ability of intruders to ship out classified information). In this experiment, routers are...
the main components of the autonomic computing environment, they can register actions in the form of Extended Access Control List (E-ACL) [6] with the CMI on the Actuator port. For example, when the online analysis modules detect the existence of abnormal network behaviors, it will report the attacker’s IP address, attacker’s port number, and victim’s IP address. The action module will then check the network topology to determine the appropriate set of routers that must be blocked in order to stop the attacks and minimize their impacts.

4 Experiments

A testbed shown in Figure 3 is set up to experiment with and evaluate the prototypical self-protection service. The testbed consists of three Cisco 2800 series routers and more than twenty computers. In this testbed, different operating systems (Linux kernels, Windows 98, Windows 2000, Windows XP and Windows 2003) are configured to emulate a comprehensive and live network environment. Some operating systems in this testbed are not patched up to date on purpose. Also, multiple network applications are configured across the testbed such as Apache and IIS Web Server, email server, DNS server, DHCP server, and FTP servers. Linux nodes include Fedora 5.0, WHAX, Backtrack, etc. To perform as a testbed environment for studying network attack detection mechanisms and network defense mechanisms, additional capabilities beyond networking are needed. As a whole, the testbed represents a structured collection of software tools and capabilities. In addition, the following software tools are used.

- Traffic load generator: It is responsible for the creation of actual end-to-end traffic flow which emulates application and/or user behaviors. The appropriate traffic type can be adjusting several parameters such as packet size, payload content, frequency, selection of network services, and other behavior characteristics. Current traffic generator can generate HTTP, FTP, TELNET, Video etc.
- Attack generator: A set of attacker tools either collected or developed in house provides a wide variety of attack mechanisms such as OS Detection; Horizontal Host Scanning; Vertical Port Scanning; TCP SYN scanning; TCP ACK scanning; TCP Connect scanning; TCP window scanning; TCP NULL scanning; TCP FIN scanning; Xmas scanning; Idle scan; IP protocol scanning; fast/slow scanning; scanning with fragment packets; scanning with set TTL; spoofing scanning; Nessus system vulnerability scanning; Nikto web server scanning; CP SYN flooding; ICMP flooding and UDP flooding.

4.1 Self-Configuration Experimental Results

The self-protection system prototype is implemented in Java. A set of XML schema has been defined to specify the management strategy for any component according to the CMI specification. The Analyzing and Planning modules are implemented using policy engines, data mining and optimization algorithms. The policy engine’s main function is to evaluate runtime conditions that become true and then formulate one or more actions that must be executed if the component requirements cannot be met using the current execution environment.

Here, we focus our analysis on evaluating the overhead associated with the autonomic framework and the effectiveness of self-protection. The first set of experiments evaluates the overhead of CRM. The system management component monitors the CPU and disk of the local host. If the CPU or disk usage exceeds a set threshold, actions (e.g., kill a process and delete files) will be taken. In the CMI being evaluated, the Monitor port has four sensor functions, the Actuator port has three actuator functions while the Policy port has two interaction policy rules and two behavior policy rules. The host is running a Linux machine (Fedora core 6, kernel 2.6.9) with Intel Xeon 2.8GHz processor, 1MB cache and 2GB RAM.

We ran single and multiple CRMs and measured CPU and memory overhead. The results shown in Table 1 indicate that the CPU overhead is very low; one CRM instance only imposes 0.2% overhead and when eight instances of CRM are running, the overhead cost is around 2.22%. Since we developed the CRM in Java, the memory overhead for one CRM is around 5.3MB and the total memory overhead is linear with respect to the number of CRMs.

4.2 Protection Experimental Results

290,872 NetFlow records are used to train the self-protection service. Among these records, there are 70,089 normal records and 220,783 abnormal records. The normal
Table 1: Overhead of CRMs running on one machine

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CRM</td>
<td>0.2%</td>
<td>5,351kB (0.26%)</td>
</tr>
<tr>
<td>4 CRMs</td>
<td>1.0%</td>
<td>21,208kB (1.05%)</td>
</tr>
<tr>
<td>8 CRMs</td>
<td>2.22%</td>
<td>45,882kB (2.28%)</td>
</tr>
</tbody>
</table>

records describe the behaviors web browsing, network time protocol, DNS, NetBIOS, file transferring, audio, video, secure shell, etc. Table 2 lists the types of attacks that were used to evaluate the protection engine. The results show that the protection engine can accurately detect and proactively protect against scanning, DoS, Worm and R2L attacks. Due to the implementation limitation of the self-protection service, it only uses the NetFlow information as the data source to build the network profile, so the self-protection service can't detect passive scanning and layer 2 scanning attacks such as Ethercap attack. Also since the current implementation does not collect and analyze the information related with host workloads and processes, it is not able to detect attacks that exploit operating system vulnerabilities (e.g., exploits such as Ownstation, snooqer, killthemessenger, smb/rpc nuke, rpc dcom, octopus, jolt2).

Table 2: Self-Protection Evaluation Results

<table>
<thead>
<tr>
<th>Attack Category</th>
<th>Attack Methods</th>
<th>Detected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanning</td>
<td>Xprobe2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>APNET</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Nikto</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Traceroute</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Whisker</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Enum</td>
<td>No</td>
</tr>
<tr>
<td>Passive Scanning</td>
<td>Ettercap</td>
<td>No</td>
</tr>
<tr>
<td>Exploits</td>
<td>Ownstation</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Snooqer</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SMB/RPC Nuke</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>RPC DCOM</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Octopus</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Jolt2</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Killthemessenger</td>
<td>No</td>
</tr>
<tr>
<td>R2L</td>
<td>Nettcat</td>
<td>No</td>
</tr>
<tr>
<td>Worm</td>
<td>Theodin worm</td>
<td>Yes</td>
</tr>
<tr>
<td>Dos Attack</td>
<td>TCP SYN flooding</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>UDP flooding</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>ICMP flooding</td>
<td>Yes</td>
</tr>
</tbody>
</table>

During the experiments, more than 20 computers were kept running various network applications that millions of NetFlow records were generated. In addition, many variations of the network attacks were launched on the testbed, the self-protection service was able to detect these attacks at a rate higher than 99% with only three false alarms reported.

The first false alarm is a UDP port 9 alarm, which is MSN 'heart beat' or the activity between one of the window host with Microsoft IP. In the training data set, there are only one normal UDP port 9 traffic records and 10 scanning records. Hence, the confidence of detection of possible attacks is 99%. The second one is during the APNET attack where legitimate nodes 10.1.1.52 and 10.1.1.53 are blocked on two particular connections. One is connection to 207.46.24.25 on port 1863 which is a suspicious MSNp protocol. The second connection is a UDP session on port 138 (Netbios - dgm) which is possible a false alarm. The third false alarm occurred during the APNET attack to block the legitimate node 10.2.1.12 from accessing http://music.yahoo.com.

5 Conclusion

In this paper, a framework to design and implement autonomic computing systems is presented. Using the autonomic computing design methodology, autonomic systems and services can be built either from scratch or by using existing legacy resources or software systems. Two software modules - Component Management Interface (CMI) and Component Runtime Manager (CRM) can be added to any legacy resource (hardware or software) to enable the autonomic computing capabilities. We have applied self-protection approach to manage our network testbed as well as its security management. The experiments conducted on the test bed showed the effectiveness of self-protection in detecting and protecting the network from a wide range of network attacks along with high detection rate accuracy and very low false alarms. We are currently evaluating the overhead of the CRM and CMI in order to build a practical autonomic system.

References


Guangzhi Qu is an assistant professor in computer science and engineering department
Osamah A. Rawashdeh, Ph.D. is an Assistant Professor in the Department of Electrical and Computer Engineering at Oakland University. He received his BS with honors, MS, and PhD in Electrical Engineering from the University of Kentucky in 2000, 2003, 2005 respectively. He absolved internships at Daimler Benz AG and at SIEMENS AG and is a member of IEEE, ACM, AIAA, and ASEE. His research interests include embedded systems design, fault-tolerance, and reconfigurable computing. Before joining Oakland University in the Fall of 2007, Dr. Rawashdeh served as a Lecturer the Department Electrical and Computer Engineering at the University of Kentucky.

Dr. Dunren Che is currently an associate professor in the Dept. of Computer Science, Southern Illinois University at Carbondale, USA. He received his PhD in Computer Science from the Beijing University of Aeronautics and Astronautics, Beijing, China in 1994, and had numerous years of post-doc research experience in several institutes. His research interests include database, data mining, bioinformatics, semantic web, cloud computing (security issues), and web application development. His recent focus is primarily on XML query processing and optimization.