ARCHITECTURE-BASED SELF-PROTECTING SOFTWARE SYSTEMS

by

Eric Yuan
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
the Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____ Dr. Hassan Gomaa, Dissertation Director

_____ Dr. Daniel A. Menascé, Committee Member

_____ Dr. Carlotta Domeniconi, Committee Member

_____ Dr. James H. Jones Jr., Committee Member

_____ Dr. Sam Malek, UC Irvine, Former Advisor

_____ Dr. Sanjeev Setia, Department Chair

_____ Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Spring Semester 2016
George Mason University
Fairfax, VA

Architecture-Based Self-Protecting Software Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Eric Yuan
Master of Science
University of Virginia, 1996
Bachelor of Science
Tsinghua University, 1993

Director: Hassan Gomaa, Professor
Department of Computer Science

Spring Semester 2016
George Mason University
Fairfax, VA

Dedication

To Grace —
Wife, Listener, Counselor, Encourager;
"but thou excellest them all" (Prov. 31:29b)

To our sons, Christopher and Geoffrey —
So this is what I've been doing after your bedtime...
You made this journey a lot more fun and bearable

Soli Deo Gloria

Acknowledgments

I would like to give my heartfelt thanks to Dr. Sam Malek, who had been my advisor all except this final semester before he moved to UC Irvine. This dissertation wouldn't have been possible without his counsel, inspiration and encouragement every step of the way. In fact, the very idea of this body of work originated from my project assignment in Sam's CS795 class. Sam was always approachable, always willing to listen, always challenging and not afraid to "call you bluff". One truly cannot ask for a better advisor.

I would like to thank my committee members, Dr. Hassan Gomaa, Dr. Daniel Menascé, Dr. Carlotta Domeniconi, and Dr. Jim Jones Jr. for their kind support and insightful feedback. Dr. Gomaa kindly accepted to be my dissertation chair after Sam's departure and provided a lot of help during the graduation process. Dr. Domeniconi's guidance on data mining techniques and literature has been a great boost to my research and is greatly appreciated. I am also indebted to the professors' flexibility to accommodate my defense out of their packed schedules.

I would also like to thank Dr. David Garlan and his team (Bradley Schmerl, Jeff Gennari et al.) from Carnegie Mellon University for the opportunity to collaborate on the QoSA'13 paper as well as their generous sharing of the Rainbow framework.

Finally, I want to thank my fellow Software Engineering PhD students Dr. Naeem Esfahani, Kyle Canavera, Dr. Ehsan Kouroshfar and Dr. Nariman Mirzaei for their help and good company. They are a sharp bunch and I wish all of them a bright future.

Table of Contents

List of Tables

List of Figures

Abstract

ARCHITECTURE-BASED SELF-PROTECTING SOFTWARE SYSTEMS

Eric Yuan, PhD

George Mason University, 2016

Dissertation Director: Dr. Hassan Gomaa

Security is increasingly a principal concern that drives the design and construction of modern software systems. Since conventional software security approaches are often manually developed and statically deployed, they are no longer sufficient against today's sophisticated and evolving cyber security threats. This has motivated the development of *self-protecting software* that is capable of detecting security threats and mitigating them through runtime adaptation techniques. Much self-protection research to-date, however, has focused on specific system layers (e.g., network, host, or middleware), but lacks the ability to understand the "whole picture" and deploy defensive strategies intelligently.

In this dissertation, I make a case for an *architecture-based self-protection* (ABSP) approach to address this challenge. In ABSP, detection and mitigation of security threats are informed by an architectural representation of the running system, maintained at runtime. With this approach, it is possible to reason about the impact of a potential security breach on the system, assess the overall security posture of the system, and achieve defense in depth.

To demonstrate the feasibility of this approach, I designed and implemented an autonomous self-protection framework called ARMOUR, consisting of three key elements: (1) a class of machine learning techniques to mine a software system's execution history at runtime, in order to develop a probabilistic architectural model that represents the system's normal behavior; (2) a novel anomaly detection algorithm that can detect in near real-time security threats that violate the system usage model; and (3) a set of architectural adaptation patterns that provide intelligent, repeatable threat mitigation against well-known web application security threats.

My extensive evaluation of the ARMOUR framework against a real world emergency response system has shown very promising results, showing the framework can effectively detect covert attacks, including insider threats, that may be easily missed by traditional intrusion detection methods, thereby providing an additional line of defense for component-based software systems. The evaluations also demonstrate many practical advantages of the framework, including unsupervised learning with no need for clean training data and potential to detect unknown threats.

Finally, I extend the ABSP approach to a deployment topology self-optimization scenario to illustrate that mining architecture-level system behavior models may help improve other quality attributes of a software system beyond security.

Chapter 1: Introduction

Security remains one of the principal concerns for modern software systems. In spite of the significant progress over the past few decades, the challenges posed by security are more prevalent than ever before. As the awareness grows of the limitations of traditional, static security models (see Section 3.1 for details), current research shifts to dynamic and adaptive approaches, where security threats are detected and mitigated at runtime, namely, *self-protection*.

Self-protection has been identified by Kephart and Chess [87] as one of the essential traits of self-management for autonomic computing systems. My systematic survey of this research area [203] shows that although existing research has made significant progress towards autonomic and adaptive security, gaps and challenges remain. Most prominantly, self-protection research to-date has primarily focused on specific line(s) of defense (e.g., network, host, or middleware) within a software system. In contrast, little research has provided holistic understanding of overall security posture and concerted defense strategies and tactics.

In this research, I would like to make a case for an *architecture-based self-protection* (ABSP) approach to address the aforementioned challenges. In ABSP, detection and mitigation of security threats are informed by an architectural representation of the software that is kept in synch with the running system. An architectural focus enables the approach to assess the overall security posture of the system and to achieve defense in depth, as opposed to point solutions that operate at the perimeters. By representing the internal dependencies among the system's constituents, ABSP provides a better platform to address challenging threats such as insider attacks. The architectural representation also allows the system to reason about the impact of a security breach on the system, which would inform the recovery process.

To prove the feasibility of the ABSP approach, I have designed and implemented an architecture-based, use case-driven framework, dubbed **AR**chitectural-level **M**ining **O**f **U**ndesired behavio**R** (ARMOUR), that involves mining software component interactions from system execution history and applying the mined architecture model to autonomously identify and mitigate potential malicious behavior.

A first step towards ABSP is the timely and accurate detection of security compromises and software vulnerabilities at runtime, which is a daunting task in its own right. To that end, the ARMOUR framework starts with monitoring component-based interaction events at runtime, and using machine learning methods to capture a set of probabilistic association rules or patterns that serve as a normal system behavior model. The framework then applies the model with an adaptive detection algorithm to efficiently identify potential malicious events. From the machine learning perspective, I identified and tailored two closely-related algorithms, Assocation Rules Mining and Generalized Sequential Pattern Mining, as the core data mining methods for the ARMOUR framework. My evaluation of both techniques against a real Emergency Deployment System (EDS) has demonstrated very promising results [55, 202, 205].

In addition to threat detection, the ARMOUR framework also calls for the autonomic assessment of the impact of potential threats on the target system and mitigation of such threats at runtime. In a recent work [206], I have showed how this approach can be achieved through (a) modeling the system using machine-understandable representations, (b) incorporating security objectives as part of the system's architectural properties that can be monitored and reasoned with, and (c) making use of autonomous computing principles and techniques to dynamically adapt the system at runtime in response to security threats, without necessarily modifying any of the individual components. Specifically, I illustrated several architecture-level *self-protection patterns* that provide reusable detection and mitigation strategies against well-known web application security threats.

My work outlined in this dissertation makes a convincing case for the hitherto overlooked role of softare architecture in software security, especially software self-protection. The

ABSP approach complements existing security mechanisms and provides additional defense-in-depth for software systems against ever-increasing security threats. By implementing self-protection as orthogonal architecture concerns, separate from application logic, this approach also allows self-protection mechanisms to evolve independently, to quickly adapt to emerging threats.

The remainder of the dissertation is organized as follows. In Chapter 2, I introduce a simple motivating example to illustrate the research challenge. In Chapter 3, I state the problem and specify the scope of my work. Before presenting the ABSP approach, I first give a more formal definition of self-protection in the context of self-adaptive systems in Chapter 4, and offer in Chapter 5 a systematic survey on the start-of-the-art self-protecting research landscape. Chapters 6 through Chapter 9 proceed to elaborate the three key elements of the ARMOUR framework: the mining of the component interaction model, the threat detection framework, and threat mitigation using architectural adaptation patterns. The insights and evaluation results from these chapters serve as strong validation of the three hypotheses from Chapter 3. In Chapter 10, I extend the ABSP approach to a deployment topology self-optimization scenario to illustrate that mining architecture-level system behavior models may help improve other quality attributes of a software system beyond security. Chapter 11 describes related research. Finally, I conclude this dissertation with the summary of the contributions, threats of validity, and future work in Chapter 12.

Chapter 2: A Motivating Example

Based on real sites like cnn.com, Znn [37] is a news service that serves multimedia news content to its customers. Architecturally, Znn is a web-based client-server system that conforms to an N-tier style. As illustrated in Figure 2.1, the service provides web browser-based access to a set of clients. To manage a high number of client requests, the site uses a load balancer to balance requests across a pool of replicated servers (two shown), the size of which can be configured to balance server utilization against service response time. For simplicity sake we assume news content is retrieved from a database and we are not concerned with how they are populated. We further assume all user requests are stateless HTTP requests and there are no transactions that span across multiple HTTP requests. This base system does not yet have any architectural adaptation features, but serves as a good starting point for our later discussions.

In the real world, a public website like Znn faces a wide variety of security threats. Malicious attacks, both external and internal, grow more complex and sophisticated by the day. The Open Web Application Security Project (OWASP), for instance, maintains a Top Ten list [133] that provides a concise summary of what was considered to be the ten most critical security risks at the application level. They are listed in Table 2.1. Another list published by the MITRE Corporation, the Top 25 Common Weakness Enumeration (CWE) [175], covers many similar threats with more details.

Mature and effective defense mechanisms are readily available to make the Znn news site more secure. To name a few:

- A traffic-filtering firewall may be placed at the WAN edge before the load balancer to filter all incoming TCP/IP traffic. Illegal access to certain ports and protocols from certain source IP addresses can be easily blocked.

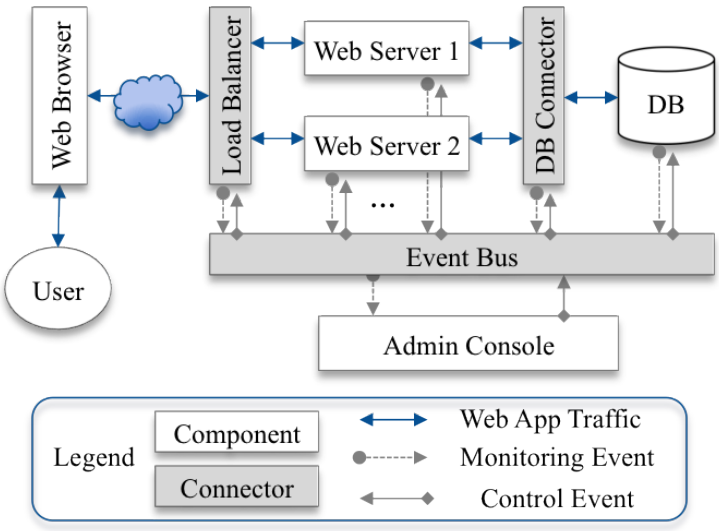


Figure 2.1: Znn Basic Architecture

- A network-based intrusion detection system (NIDS) device may be installed within the local area network, behind the firewall, that examines network traffic to detect abnormal patterns. Repeated requests to a single server that exceed a threshold, for example, may trigger an alarm for Denial of Service (DoS) attack.

- A host-based intrusion detection system (HIDS) in the form of software agents can be deployed on all servers, which monitors system calls, application logs, and other resources. Access to a system password file without administrator permissions, for instance, may indicate the server has been compromised.

- The Admin Console can define and manage Access Control Lists (ACL) for the web application. Only users with sufficient privileges may be allowed to perform certain operations, such as writing to the database.

"Offline" approaches are also available during the software development lifecycle to make the Znn application less prone to security attacks. For example, OWASP has also recommended a rich set of detection techniques and prevention countermeasures aimed at mitigating the aforementioned Top 10 risks. Some key practices include [133]:

Table 2.1: OWASP Top 10, 2010 Edition

A1	Injection
A2	Cross-Site Scripting (XSS)
A3	Broken Authentication and Session Management
A4	Insecure Direct Object References
A5	Cross-Site Request Forgery (CSRF)
A6	Security Misconfiguration
A7	Insecure Cryptographic Storage
A8	Failure to Restrict URL Access
A9	Insufficient Transport Layer Protection
A10	Unvalidated Redirects and Forwards

- Conduct code reviews and blackbox penetration testing to find security flaws proactively;
- Employ static and dynamic program analysis tools to identify application vulnerabilities. For SQL injection risks, for instance, one can conduct static code analysis to find all occurrence of the use of SQL statement interpreters;
- Use whitelist input validation to ensure all special characters in form inputs are properly escaped to prevent Cross-Site Scripting (XSS) attacks;
- Follow good coding practices and use well-tested API libraries.

The traditional approaches, however, are not without limitations. In the next chapter I will highlight some of them as I define the research problem and formulate the research hypotheses.

We will also return to the Znn example later in the thesis to illustrate how architecture-based approaches may be utilized to help counter such threats in an autonomous and self-managed fashion. Note that we focus our attention on the OWASP Top 10 and CWE Top 25 threats due to their prevalence on the Internet and their relevance to the Znn web application. However the same general approach can be applied to counter other security threats not covered in the two lists, such as Denial of Service attacks, buffer overflows, privilege escalation, etc.

Chapter 3: Research Problem

In this chapter, I describe the motivating factors behind my research and proceed to define the research problem and formulate the specific hypotheses that this thesis will seek to investigate and validate.

3.1 Problem Statement

The traditional security approaches, some of which mentioned in the previous chapter, are effective but not without limitations.

First, most are labor-intensive and require significant manual effort during development and at runtime. Second, the testing tools and preventive countermeasures do not provide complete and accurate defense against rapidly changing attack vectors, as admitted in the OWASP Top 10 report. Many approaches find it difficult to balance the competing goals of maximizing detection coverage (reducing false negatives) and maintaining detection accuracy (reducing false positives). Furthermore, for common web attacks such as XSS and CSRF, they are partly caused by careless and unwitting user behavior, which is impossible to completely eliminate. Last but not the least, a large percentage of the web applications today are so-called "legacy" applications for which development had ended some time ago. Even when vulnerabilities are found, the fixes are going to be difficult and costly to implement.

There are two alarming trends that are making the problem worse and forcing many software systems to be on the defensive:

1. *The growing sophistication, agility, and persistence of cyberattacks.* As software systems become more networked, interactive and ubiquitous, they are more prone to

malicious attacks. Over the years the frequency, complexity, and sophistication of attacks are rapidly increasing, causing severe disruptions of online systems with sometimes catastrophic consequences. From some of the well-publicized recent incidents we can get a glimpse of the characteristics of such threats. The Conficker worm, first detected in 2008, caused the largest known computer infection in history and was able to assemble a botnet of several million hosts – an attack network that, if activated, would be capable of large-scale Distributed Denial of Service (DDoS) attacks. What is unique about Conficker is not just the scale it achieved, but also its use of sophisticated software techniques including peer-to-peer networking, self-defense through adaptation, and advanced cryptography [49]. The Stuxnet worm, discovered in 2010, is the first known malware to target and subvert industrial control systems. In addition to being credited with damaging the Iranian nuclear program, the attack demonstrates its ability to target multiple architecture layers of the target system – exploiting the network and host-level vulnerabilities is only a stepping stone for malicious actions at the application level [99]. Likewise the Duqu worm, discovered in September 2011, is a reconnaissance worm that does no harm to the infected systems but is tasked to collect and exfiltrate information such as valid digital certificates that may be used in future attacks. It further illustrates the deliberate, coordinated and persistent nature of today's cyber threats [17].

2. *The rise of insider attacks.* Users with legitimate credentials can do almost anything they want with the systems they have access to, with very little control or oversight. As such, consequences from malicious insider incidents can be substantial and often more devastating than external attacks, especially for critical sectors such as banking, critical infrastructure, and national security [25]. Unfortunately, most existing security countermeasures are focused on protecting the system at its boundary and thus ineffective against insider threats.

Under such growing threats, conventional and often manually developed and statically

employed techniques for securing software systems are no longer sufficient. This has motivated the development of *self-protecting software*—a new kind of software, capable of detecting security threats and mitigating them through runtime adaptation techniques.

Self-protection has been identified as one of the essential traits of self-management for autonomic computing systems. Kephart and Chess characterized self-protection from two perspectives [87]: From a reactive perspective, the system automatically defends against malicious attacks or cascading failures, while from a proactive perspective, the system anticipates security problems in the future and takes steps to mitigate them. Self-protection is closely related to the other self-* properties, such as self-configuration and self-optimization. On one hand, a self-configuring and self-optimizing system relies on self-protection functions to ensure the system security remains intact during dynamic changes. On the other hand, the implementation of self-protection functions may also leverage the same techniques used for system reconfiguration and optimization.

As will be made evident in Chapter 5, most self-protection research to-date has focused on specific line(s) of defense (e.g., network, host, or middleware) within a software system. Specifically, such approaches tend to focus on a specific type or category of threats, implement a single strategy or technique, and/or protect a particular component or layer of the system [204]. As a result, the "big picture" understanding of overall security posture and globally coordinated defense strategies appear to be lacking. In addition, growing threats of insider attack call for new mechanisms to complement traditional perimeter-based security (i.e., securing the system at its boundaries) that has been the main focus of prior research. Finally, due to the added complexity of dynamically detecting and mitigating security threats, the construction of self-protecting software systems has shown to be significantly more challenging than traditional software [38]. Lack of engineering principles and repeatable methods for the construction of such software systems has been a major hindrance to their realization and adoption by industry.

In this thesis, I make a case for the critical yet often overlooked role of *software architecture* in building self-protecting software systems. As introduced in Chapter 1, I propose

the ABSP approach to address the aforementioned challenges. ABSP focuses on securing the architecture as a whole, as opposed to specific components such as networks or servers. Working primarily with constructs such as components, connectors and architecture styles, the ABSP approach protects the system against security threats by (a) modeling the system using machine-understandable representations, (b) incorporating security objectives as part of the system's architectural properties that can be monitored and reasoned with, and (c) making use of autonomous computing principles and techniques to dynamically adapt the system at runtime in response to security threats, without necessarily modifying any of the individual components. As such, the ABSP approach does not seek to replace the mainstream security approaches but rather to complement them.

In summary, my thesis seeks to **(1) provide concrete evidence on how architecture plays a vital role in engineering self-protecting software systems; (2) develop a disciplined and repeatable framework that utilizes architecture models to detect and respond to malicious threats dynamically; and (3) provide empirical evidence that the ABSP framework enhances the overall security of real-world software systems.**

3.2 Research Hypotheses

My research starts with a thorough survey of the state-of-the-art self-protection research, before investigating the following overarching hypotheses:

Hypothesis 1: It is possible to mine a probabilistic architectural model that captures a software system's normal behavior from its execution history.

Because statically built architecture models tend to get out of sync with the real system due to adaptation and architecture decay, the architecture-based detection models must be built and maintained *at runtime* in order to counter emerging and often unanticipated attacks. I hypothesize that an alternative approach can be developed that does not require

defining the system's behavior model beforehand, but instead employ modern data mining and machine learning techniques to "discover" an architecture representation of the system from its execution history for self-protecting purposes.

Hypothesis 2: Using an automatically mined architectural model, it is possible to detect a large-class of security threats that violate the normal usage of the software.

A first step towards ABSP is the timely and accurate detection of security compromises and software vulnerabilities at runtime. I hypothesize that, because architecture models provide a "big picture" view of the entire system, they may be used to reason about the overall security posture of the system and detect potentially malicious behabvior that may otherwise go unnoticed by traditional detection mechanisms that only focus on a specific aspect of the system (such as its networks or hosting operating systems).

Hypothesis 3: Software architecture provides an additional line of defense for effectively mitigating security threats at runtime.

I hypothesize that with the ABSP approach, it is possible to reason about the impact of a potential security breach on the system, assess the overall security posture of the system, and use repeatable architecture adaptation techniques to mitigate well-known application security threats.

Chapter 4: Self-Protection Defined

As a starting point, it is important to establish a working definition of self-protection property, given that our experience shows the term has been used rather loosely in the literature. The goal of this definition is to clarify *what* we consider to be a self-protecting software system, which in turn defines the scope of this thesis. Our understanding of the self-protection property is consistent with that laid out in FORMS [191], a formal language for specifying the properties and architectures of self-* (i.e., self-management, self-healing, self-configuration, and self-protection) software systems. According to FORMS, a software system exhibiting a self-* property is comprised of two subsystems: a meta-level subsystem concerned with the self-* property that manages a base-level subsystem concerned with the domain functionality.

Figure 4.1 shows what I consider to be a self-protecting software system in light of FORMS concepts. The meta-level subsystem is part of the software that is responsible for protecting (i.e., securing) the base-level subsystem. The meta-level subsystem would be organized in the form of feedback control loop, such as the Monitor, Analysis, Planning, Execution, and Knowledge (MAPE-K) architecture [87] depicted in the figure.

As an illustrating example, I now modify the Znn news service introduced in Chapter 2 to follow this reference architecture. Instead of manually managing system security via the Admin Console, I augment Znn's basic web application architecture with a "meta component" called Architecture Manager (AM), shown in Figure 4.2. The AM maintains an up-to-date model of the system's architecture properties, and is responsible for monitoring the components and connectors in the system and adapting them in accordance with self-protection goals. To achieve this, the AM implements the aforementioned MAPE-K loop, and is connected to "sensors" throughout the system to monitor their current status and

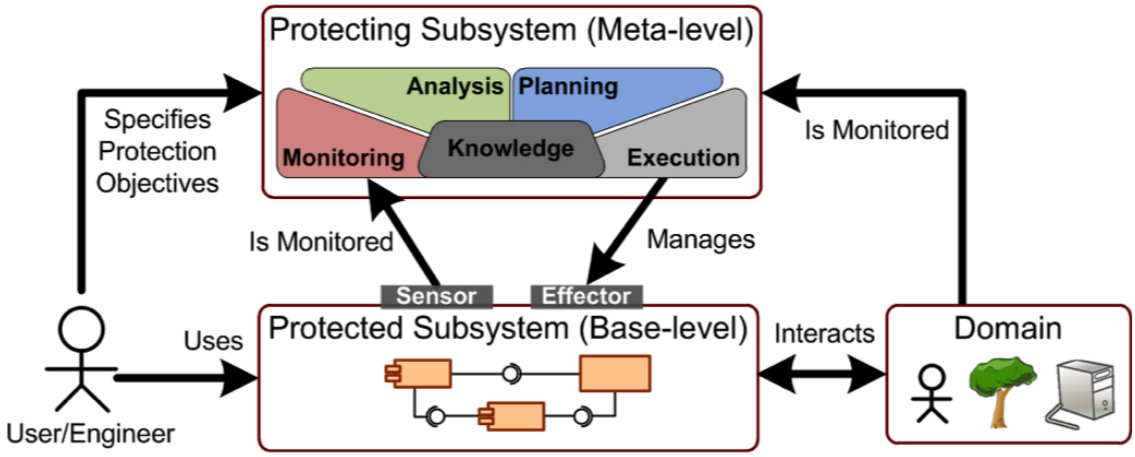


Figure 4.1: Self-Protection in Light of FORMS Reference Architecture

"effectors" that can receive adaptation commands to make architecture changes to the base system at runtime.

One should not interpret this reference architecture to mean that the base level subsystem is agnostic to security concerns. In fact, the base-level subsystem may incorporate various security mechanisms, such as authentication, encryption, etc. It is only that the decision of when and how those security mechanisms are employed that rests with the meta-level subsystem.

In addition to the intricate relationship between the meta-level and base-level subsystems, we make two additional observations. First, we underline the role of humans in such systems. Security objectives often have to be specified by human stakeholders, which are either the systems users or engineers. The objectives can take on many different forms (e.g., access control policies, anomaly thresholds). Second, we observe that for self-protection to be effective, it needs to be able to observe the domain environment within which the software executes. The domain environment is comprised of elements that could have an impact on the base-level software, but are outside the realm of control exercised by the meta-level subsystem. For instance, in the case of an online banking system, the domain could be other banking systems, which could impact the security of the protected system,

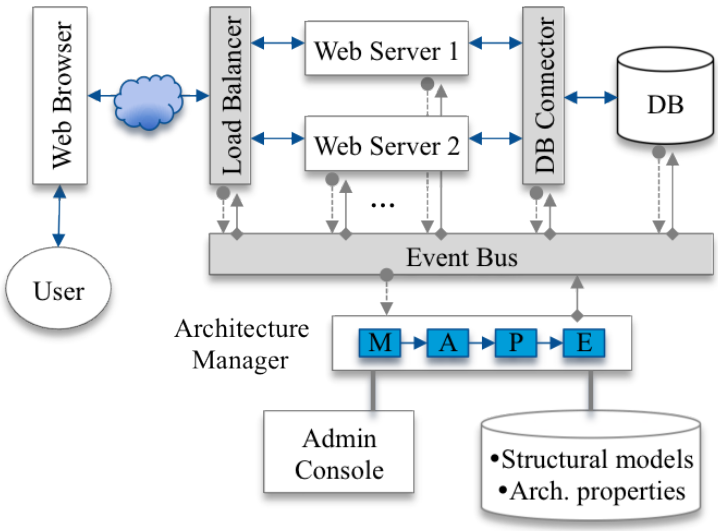


Figure 4.2: Znn Self-Protection Architecture

but the meta-level subsystem has no control over them.

These concepts, although intuitive, have allowed me to define the scope of my research. For instance, we were able to distinguish between an authentication algorithm that periodically changes the key it uses for verifying the identities of users, and a system that periodically changes the authentication algorithm it uses at runtime. The former we classify to be an *adaptive security* algorithm, as the software used in provisioning security does not change, and therefore outside the scope of this thesis. Whereas the latter we classify to be a self-protecting software system, as it changes the software elements used in provisioning security, and therefore within the scope of my research. Though other reference frameworks exist (such as control theory-based DYNAMICO [185]), I will use the basic concepts introduced in this chapter for the rest of the thesis to illustrate the differences between the self-protection approaches surveyed.

Chapter 5: A Systematic Survey of Self-Protecting Software Systems

In this chapter, I start with a systematic survey to understand the start of the art of self-protecting research approaches and techniques. As Section 11.1 testifies, no such surveys had been conducted before that can provide a comprehensive view for our research.

5.1 Survey Objectives

Under the reference architecture introduced in Chapter 4, the "meta-level" subsystem may be used to monitor and mitigate security threats to the base-level subsystem in more sophisticated ways. Consider the self-protecting Znn system depicted in Figure 4.2, for example, the Architecture Manager (AM) may thwart an intruder to the site in the following ways:

- Upon sensing an unusual data retrieval pattern from the Windows server, the AM shuts down the server and redirects all requests to a backup server accordingly;
- Alternatively, the AM could monitor and compare the behavior from all web server instances. The AM may detect an anomaly from the compromised web server instance (e.g., using a majority voting scheme) and consequently shuts it down.

Compared with the more conventional security countermeasures described in Chapter 2, the two examples above clearly exhibit self-adaptive and self-protecting behavior at the system level. As this chapter will show later, many other self-protecting mechanisms are possible. How do these different approaches compare against one another? Are some more effective than others? If so, under what conditions? To better answer these questions, one

must methodically evaluate the state of the art of the self-protection approaches, architectures, and techniques, and assess how they address the externally-driven and internally-driven security needs mentioned above.

To achieve this goal, I have taken two key approaches. First, I followed a systematic literature review process proposed by Kitchenham [90]. The process ensures the broad scope of the survey and strengthens the validity of its conclusions. Section 5.2 describes the research methodology in details. Second, I proposed a comprehensive taxonomy for consistently and comprehensively classifying self-protection mechanisms and research approaches. The taxonomy provides a coherent point of reference for all the surveyed papers and helps to identify patterns, trends, gaps, and opportunities. The taxonomy is introduced in Section 5.3. Section 5.4 classifies current and past self-protection research initiatives against the proposed taxonomy, and offers the detailed observations and analysis.

5.2 Research Method

This survey follows the general guidelines for systematic literature review (SLR) process proposed by Kitchenham [90]. We have also taken into account the lessons from Brereton et al. [22] on applying SLR to the software engineering domain. The process includes three main phases: planning, conducting, and reporting the review. Based on the guidelines, we have formulated the following research questions, which serve as the basis for the systematic literature review:

1. **RQ1**: How can existing research on self-protecting software systems be classified?
2. **RQ2**: What is the current state of self-protection research w.r.t. this classification?
3. **RQ3**: What patterns, gaps, and challenges could be inferred from the current research efforts that will inform future research?

To answer these research questions, I have developed the methodology and task flow that are described in section 5.2.1, as well as the detailed SLR protocol including key

words, sources, and selection criteria, described in section 5.2.2. Out of this process, we have included 107 papers published from 1991 to 2013, out of the total of over 1037 papers found.

No survey can be entirely comprehensive. Our keywords-based search protocol restricts us to papers that explicitly address the self-protection topic while potentially leaving out relevant papers under different terms. The limited number of keyword phrases we used (see Section 5.2.2 below) in order to keep the study manageable may have prevented the coverage of some interesting areas, such as:

- research pertaining to security assurance and security Service Level Agreements (SLA) during autonomic adaptation of services/components;
- risk-based self-protecting solutions;
- adaptive solutions addressing privacy.

5.2.1 Research Tasks

To answer the three research questions, we organized our tasks into a process flow tailored to our specific objectives, yet still adhering to the three-phase SLR process. The overall process flow is outlined in Figure 5.1 and briefly described here.

First, in the planning phase, I defined the review protocol that includes selection of the search engines, the initial selection of the keywords pertaining to self-protecting software systems, and the inclusion/exclusion criteria for the candidate papers. The protocol is described in detail in section 5.2.2.

The initial keyword-based selection of the papers is an iterative process that involves exporting the candidate papers to a "research catalog" and applying the pre-defined inclusion/exclusion criteria on them. In the process, the keyword search expressions and the inclusion/exclusion criteria themselves may also need to be fine-tuned, which would in turn trigger new searches. Once the review protocol and the resulting paper collection were stabilized, our research team also conducted peer-reviews to validate the selections.

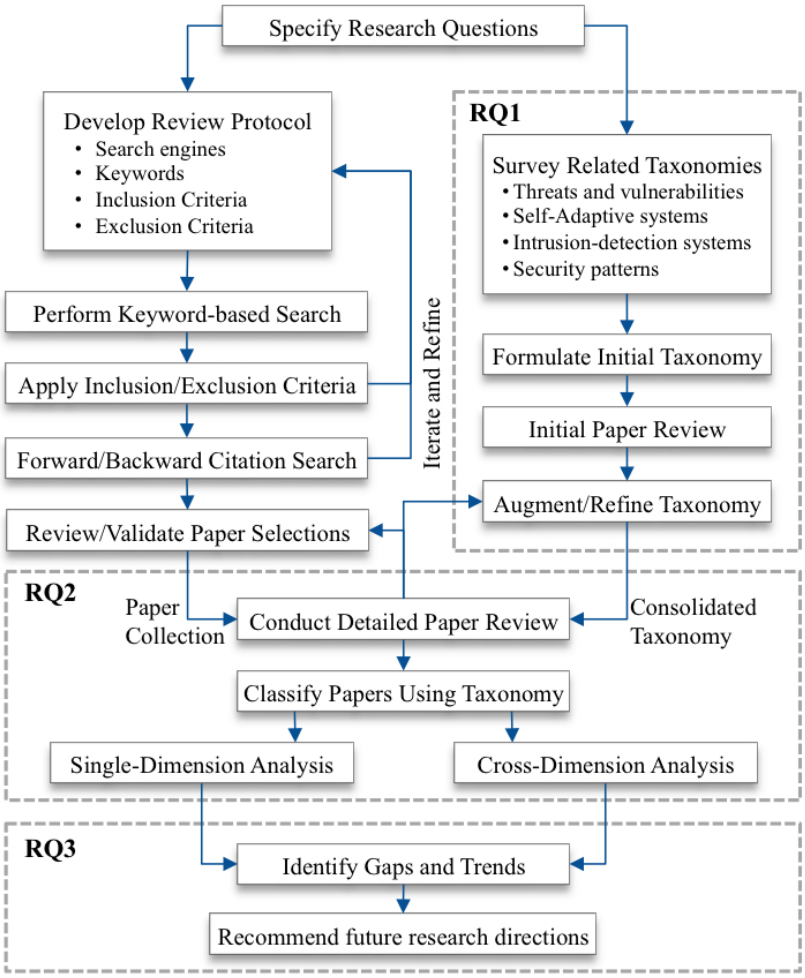


Figure 5.1: Survey Process Flow and Tasks

For RQ1, in order to define a comprehensive taxonomy suitable for classifying self-protection research, I first started with a quick "survey of surveys" on related taxonomies. Since our research topic straddles both the autonomic/adaptive systems and computer security domains, I identified some classification schemes and taxonomies from both domains, as described in Section 11.1. After an initial taxonomy was formulated, I then used the initial paper review process (focusing on abstracts, introduction, contribution, and conclusions sections) to identify new concepts and approaches to augment and refine our taxonomy. The resulting taxonomy is presented in Section 5.3.

For the second research question (RQ2), I used the validated paper collection and the consolidated taxonomy to conduct a more detailed review on the papers. Each paper was classified using every dimension in the taxonomy, and the results were again captured in the research catalog. The catalog, consisting of a set of spreadsheets, allowed us to perform qualitative and quantitative analysis not only in a single dimension, but also across different dimensions in the taxonomy. The analysis and findings are documented in Section 5.4.

To answer the third research question (RQ3), I analyzed the results from RQ2 and attempted to identify the gaps and trends, again using the taxonomy as a critical aid. The possible research directions are henceforth identified and presented in Section 5.5.

5.2.2 Literature Review Protocol

The first part of our review protocol was concerned with the selection of search engines. As correctly pointed out in [22], no single search engine in the software engineering domain is sufficient to find all of the primary studies, therefore multiple search engines are needed. I selected the following search sites to have a broad coverage: IEEE Explore, ACM Digital Library, Springer Digital Library, Elsevier ScienceDirect (Computer Science collection), and Google Scholar.

For these search engines I used a targeted set of keywords, including: Software Self-Protection, Self-Protecting Software, Self-Securing Software, Adaptive Security, and Autonomous Security. It is worth noting that the exact search expression had to be fine-tuned for each search engine due to its unique search interface (e.g. basic search vs. advanced search screens, the use of double quotes, and the AND/OR expressions). In each case I tried to broaden the search as much as possible while maintaining a manageable result set. For example, because Google Scholar invariably returns thousands of hits, we limited our search to the first 200 results. I also used the following inclusion and exclusion criteria to further filter the candidate papers:

- Only refereed journal and conference publications were included.
- Based on our definition of self-protection in Chapter 4, I included autonomic and

adaptive software research that is directly relevant to the self-protecting and self-securing properties. Other properties such as self-healing and self-optimization are out of the scope of this survey.

- Our review focuses on software systems only, therefore does not include self-protection pertaining to hardware systems.
- Software security research that doesn't exhibit any self-adaptive/autonomic traits is excluded.
- Our definition of self-protection pertains to protecting the software system against malicious threats and attacks. Sometimes other connotations of self-protection may be possible. For example, protecting a system from entering into an inconsistent state (from a version consistency perspective), or protecting a wireless sensor network (from the field-of-view perspective) may also be viewed as self-protection. Such papers are excluded in this review.
- Position papers or research proposals not yet implemented or evaluated are excluded.

When reviewing a candidate paper, I have in many occasions further extended the collection with additional papers that appear in its citations or those that are citing it (backward and forward citation search).

5.3 Taxonomy

To define a self-protection taxonomy for RQ1, I started with selecting suitable dimensions and properties found in existing surveys. The existing taxonomies described in Section 11.1, though relevant and useful, are not sufficiently specific and systematic enough for classifying self-protection approaches in that they either focus on adaptive systems in general, but not specifically on security, or focus on software security in general, but not on autonomic and adaptive security. Many focus on only certain architectural layers of software systems (such as middleware). Even when a taxonomy dimension is appropriate for our purposes here,

it is oftentimes too generic (e.g., open vs. closed) and need to be further qualified in the self-protection context.

Furthermore, many of the taxonomies and classification schemes lean heavily towards implementation tactics and techniques (such as those for implementation patterns) but perhaps fall short on covering architectural strategies or styles (though some exceptions do exist, such as [126]).

For such reasons, I have defined our own taxonomy to help classify existing self-protection and adaptive security research. The proposed taxonomy builds upon the existing ones surveyed earlier, and is a refinement and substantial extension of what I proposed in earlier work [204]. It consists of 14 dimensions that fall into three groups: Approach Positioning, Approach Characterization, and Approach Quality. They are defined and illustrated in Figure 3 and Figure 4, and explained in the following subsections.

5.3.1 Approach Positioning

The first part of the taxonomy, Approach Positioning, helps characterize the "WHAT" aspects, that is, the objectives and intent of self-protection research. It includes five dimensions, as depicted in the left part of Figure 5.2:

(T1) Self-Protection Levels

This dimension classifies self-protection research based on the level of sophistication of its meta-level subsystem (as defined in Chapter 4). "Monitor & Detect" is the most basic level, indicating the protecting subsystem is equipped with the capability to constantly monitor for security threats and detect anomalous or harmful activities from normal system operations. The next level is "Respond & Protect", which indicates the subsystem's ability to take action against the detected attack or anomaly. This implies the protecting subsystem can, ideally in an autonomous fashion, (a) characterize and understand the nature/type of the attacks, and (b) deploy the proper countermeasures to mitigate the security threat and maintain normal system operations to the extent possible a property often called "graceful degradation".

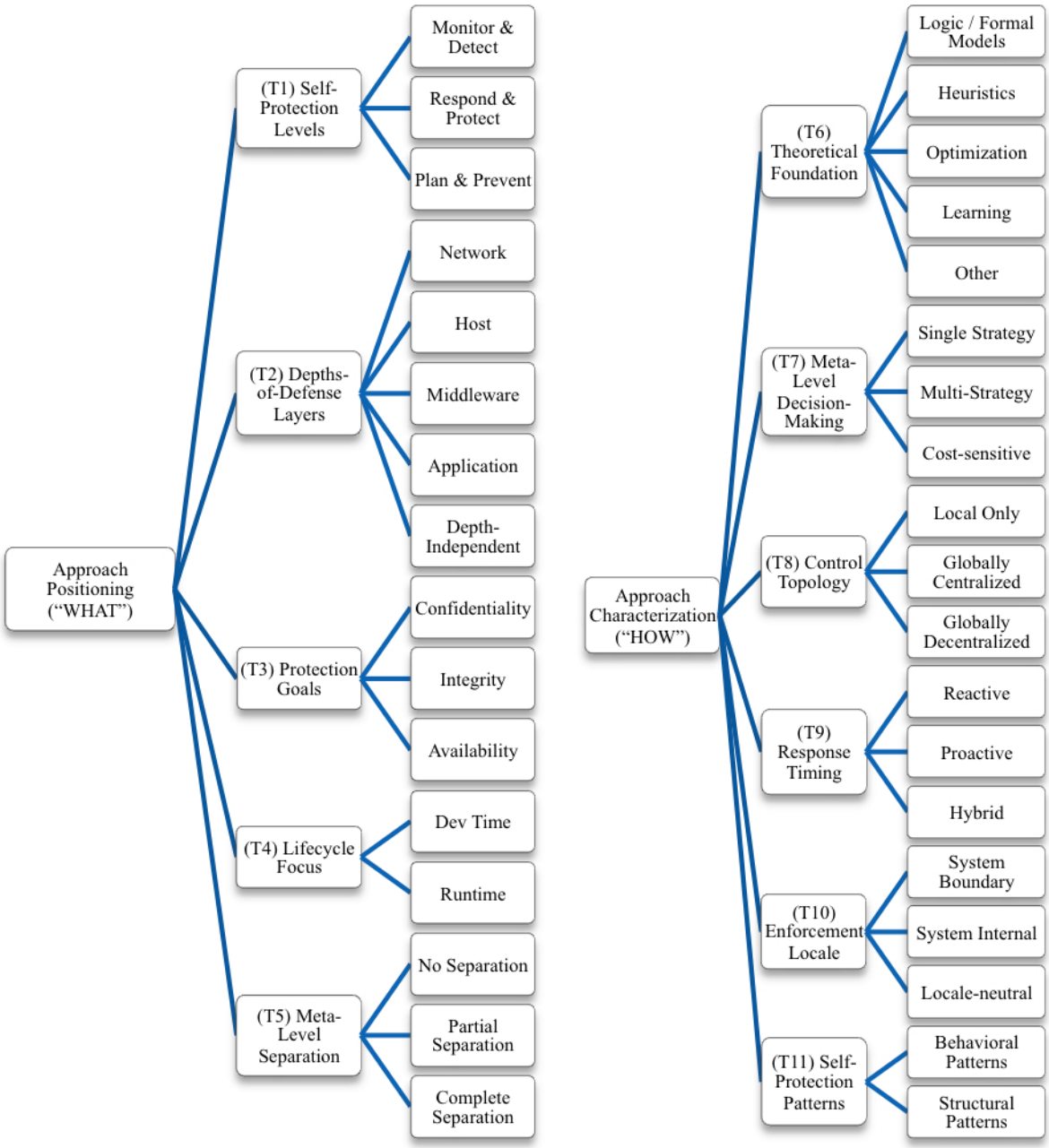


Figure 5.2: Proposed Taxonomy for Self-Protection Research

The third level, "Plan & Prevent", represents the highest level of sophistication; a security approach reaching this level allows a system to adapt and strengthen its security posture based on past events so that known security faults are prevented. I illustrate this dimension using the Znn example:

- The system is at the *Monitor & Detect* level if it is equipped with a network based IDS device connected to the router, which can detect an intrusion attempt based on known attack signatures (such as a DoS attack to the banking server), and generate an appropriate alert to the AM, which acts as the meta-level subsystem for self-protection;
- The system is at the *Respond & Protect* level if, in addition to the previous level, the AM layer responds to the router alert and changes the firewall policy to block all traffic from the source domain;
- The system is at the *Plan & Prevent* level if, in addition to the previous level, the AM also reviews the history of such attacks and moves the web server to a different IP address, so that future DoS attacks are rendered ineffective.

The three levels are consistent with (and in fact inspired by) Kramer and Magee's three-level reference architecture for self-managed systems [93]. It is easy to see the mapping from the self-protection levels to Component Management, Change Management, and Goal Management, respectively. It may also be envisioned that each self-protection level may have its own MAPE-K loop; therefore this dimension is not in conflict with the IBM reference architecture.

(T2) Depths-of-Defense Layers

This dimension captures the renowned security principle of Defense in Depth, which simply acknowledges the fact that no single security countermeasure is perfectly effective and multiple layers of security protections should be placed throughout the system. For self-protecting systems, the following defensive layers are possible, starting with the outmost

layer (please see Section 5.4.1 for examples of security countermeasures at each layer):

- *Network*: Focuses on communication links, networking protocols, and data packets.
- *Host*: Involves the host environment on a machine, involving hardware/firmware, OS, and in some occasions hypervisors that support virtual machines.
- *Middleware*: With the prevalence of component-based and service-oriented systems, use of middleware such as application servers (e.g. JEE), object brokers (e.g. CORBA) and service buses (e.g. JBoss ESB) are becoming a common practice and as such may be used as an additional layer of defense.
- *Applications*: As the last line of defense, application level security is usually concerned with programming language security and application-specific measures.
- *Depth-Independent*: This fifth layer is a special value to indicate any self-protection research that is not specific to any architecture layers. One example may be an approach that addresses self-protection in terms of software architecture abstractions such as software components, connectors, configurations, and architecture styles. A software architecture-based approach enjoys many benefits such as generality, abstraction, and potential for scalability, as pointed out by Kramer and Magee [93].

The counter-intrusion example given earlier for dimension **T1** is clearly a network layer defense. The online banking system can also choose to have a host-level defense such as a periodic patching mechanism to install OS patches that remediate Windows OS vulnerabilities, a middleware level defense that configures a cluster of redundant application servers under a Byzantine agreement protocol (as described by Castro and Liskov [30]), and an application-level defense where the AM component dynamically directs the web-based banking application to adopt different levels of security policies.

(T3) Protection Goals

This dimension classifies research approaches according to the security goal(s) they intend to achieve. Here we follow the traditional CIA model for its simplicity:

- *Confidentiality*: to protect against illegal access, spoofing, impersonation, etc.
- *Integrity*: to protect against system tampering, hijacking, defacing, and subversion
- *Availability*: to protect against degradation or denial of service

Other goals such as Accountability, Authenticity and Non-Repudiation may also be considered as implicit sub-goals that fit under this model.

In some cases a security countermeasure may help meet multiple protection goals. Suppose in the online banking example, the banking application is compiled using the StackGuard compiler [Cowan et al. 1998] to safeguard against buffer overflow attacks. This technique stops the intruder from obtaining user financial data stored in memory (confidentiality) and from illegally gaining control of the banking application (integrity) through buffer overflows. Note that in this case the technique does not help with the availability goal; I will return to this point later in the chapter.

(T4) Lifecycle Focus

This dimension indicates what part of the software development lifecycle (SDLC) a self-protection approach is concerned with. For the purposes of this thesis I simply use two phases, *Development Time* and *Runtime*, with the former encompassing also the design, testing, and deployment activities. Security at runtime is undoubtedly the primary concern of self-protection. Nonetheless, from the software engineering viewpoint it is also necessary to take into account how to better design, develop, test, and deploy software systems for self-protection.

As a concrete example, suppose all runtime system auditing data is made available to the development team of the online banking system. By feeding data into the automated testing

process, the team can make sure all new code is regression-tested against previously known vulnerabilities, or use the system logs to train the meta-level self-protection mechanisms.

(T5) Meta-Level Separation

This dimension indicates how Separation of Concerns as an architectural principle is applied in a self-protection approach. Specifically, the FORMS reference architecture [191] calls for the separation between the meta-level subsystem and the base-level subsystem, logically and/or physically. The degree of separation is useful as a telltale sign of the degree of autonomicity of the system. In the security context it also takes on an added significance, as the meta-level self-protection logic often becomes a high value target for the adversary and thus needs special fortification. Three values are used here *No Separation*, *Partial Separation*, and *Complete Separation*. Continuing with the Znn system example, complete separation of security concerns is achieved when all self-protection logic is contained in the AM component as illustrated in Figure 4.2, and the component (along with communication channels to/from it) is deployed in dedicated, trusted hardware. On the other hand, if the AM runs in the same server as the Znn application itself, or the security policy decisions are embedded in the Znn web application code, the degree of separation is low.

5.3.2 Approach Characterization

The second group of the taxonomy dimensions are concerned with classifying the "HOW" aspects of self-protection research. It includes five dimensions shown in the right half of Figure 5.2:

(T6) Theoretical Foundation

As a self-protecting software system takes autonomic and adaptive actions against malicious attacks, it often needs to consider many factors from the runtime environment and choose the optimal or near-optimal course of action out of a vast problem space. The theoretical foundation of the approach, as captured in this dimension, is therefore critical and deserves

close examination. The following sub-categories are defined:

- *Logic / formal models*, which involve logic or other mathematically based techniques for defining security related properties, as well as the implementation and verification of these properties. The design of the online banking system, for example, may include security policies formulated by finite state automata (such as those defined by Schneider [155]), and formal proof of policy enforceability;

- *Heuristics based*, which include knowledge-based, policy-based, or rule-based models whose parameters may change at runtime. For example, the online banking system may implement a policy that disables a user account when suspicious fund withdrawal patterns arise. In this case these patterns are based on heuristic rules such as a maximum daily withdrawal threshold. The system may further lower or increase the threshold according to security threat levels;

- *Optimization*, which employs analytical techniques that model security-related system behavior through quantitative metrics that is used to select the optimal adaptation strategy. For example, the banking system may use a utility function to model a user's preference between convenience/user-friendliness and strengths of protection, and set security policies accordingly (e.g. username/password vs. multi-factor authentication);

- *Learning based models*, including a rich variety of techniques that use historical or simulated data sets to train the system's autonomic defences. The learning process could be based on cognitive, data mining, stochastic/probabilistic models, etc. The Znn systems router, for instance, may use a neural net algorithm to differentiate intrusions from normal network behavior [95]).

Note that these models are not meant to be mutually exclusive. In fact, as will be seen in the survey results, many approaches leverage more than one model.

(T7) Meta-Level Decision-Making

This dimension attempts to further characterize self-protection research by examining its decision-making strategy and thought process. Here we adopt the following rather coarsely grained values (again, illustrated using the online banking system example):

- *Single strategy*, the simplest approach with a single objective, a single decision model, or a single type of attacks/vulnerabilities in mind (many examples given earlier fall into this category).

- *Multi-strategy*, involving multiple levels of decisions, metrics, and tactics. For instance, consider a situation in which the Znn system deploys two intrusion detection sensors, one network-based and the other host-based on the application server. Simple intrusions such as port scanning and buffer overflows are deterred at the device level, but the AM component may also want to correlate the network and host alerts to look for higher-level attack sequences.

- *Cost-sensitive modeling*, a special case in which security decisions involve trade-offs with other non-security related factors, such as costs or Quality of Service (QoS) requirements. For example, under certain situations, the banking application may not shut itself down to cope with a user account breach because many other legitimate users will be impacted, resulting in big loss of revenue.

(T8) Control Topology

More often than not, modern software-intensive systems are logically decomposed into separate self-contained components and physically deployed in distributed and networked environments. Self-protection functionality, therefore, needs to be implemented and coordinated among different components and machines. This dimension looks at whether a self-protection approach focuses on controlling the *local* (i.e., a single host or node) or *global* scale of the system. For those approaches at the global scale, this dimension also specifies whether they use centralized or decentralized coordination and planning. Under a

centralized topology, system self-protection is controlled by a single component that acts as the "brain", whereas under a decentralized topology, the nodes often "federate" with one another in a peer-to-peer fashion without relying on a central node. In the Znn news system, for instance, self-protection is globally centralized if the AM component is hosted on a dedicated server that monitors, controls, and adapts security countermeasures on all other devices and servers. Alternatively, if the system consists of multiple interconnected news servers (possibly at different locations) and each server hosts its own architecture manager component, the topology is globally decentralized. In a more trivial situation, the topology would be "local only" if the self-protection technique is used within a single server.

(T9) Response Timing

This dimension indicates when and how often self-protecting actions are executed, which in turn is dependent on whether the approach is reactive or proactive. In *reactive* mode, these actions occur in response to detected threats. In *proactive* mode, they may occur according to a predefined schedule, with or without detected threats. Some systems may include both modes. The security countermeasures illustrated earlier using the Znn example, such as intrusion prevention or controlling access to a user account, all fall into the reactive category. Alternatively, the Znn system could use software rejuvenation techniques (introduced by Huang et al. [80]) to periodically restart the web application instances to a pristine state, to limit damage from undetected attacks.

(T10) Enforcement Locale

This dimension indicates where in the entire system self-protection is enforced. Here we adopt a metric from [70] and define the values as *System Boundary* or *System Internal*. In the former case, self-protection is enforced at the outside perimeter of the system (such as firewalls, network devices, or hosts accessible from external IP addresses). In the latter case, self-protection mechanisms cover internal system components. The distinction may be easily seen in the Znn example: The router and the firewall represent the system boundary

that needs to be protected against intrusions, whereas the web server and the database components represent system internals that may also be protected by access control policies issued from the AM component. Self-protection approaches independent of enforcement locations are categorized as *locale-neutral*.

(T11) Self-Protection Patterns

This dimension indicates any recurring architectural patterns that rise from the self-protection approaches. Many architecture and design patterns exist, but as we can see in the next section several interesting patterns have emerged in our research as being especially effective in establishing self-protecting behavior. Here I simply mention them in two groupings and describe their details in Section 5.4.7

- *Structural* patterns that use certain architectural layouts to situate, connect, and possibly reconnect system components to achieve better integrity, robustness, and resiliency against attacks.
- Structural patterns that seek to reconfigure and adapt the runtime behavior of existing system components and their connections without necessarily changing the system architecture.

Please note that these patterns are not mutually exclusive. It is conceivable that a system may use a combination any number of them to provide more vigorous and flexible self-protection behavior.

5.3.3 Approach Quality

The third and last section of the taxonomy is concerned with the evaluation of self-protection research. Dimensions in this group, as depicted in Figure 5.3, provide the means to assess the quality of research efforts included in the survey.

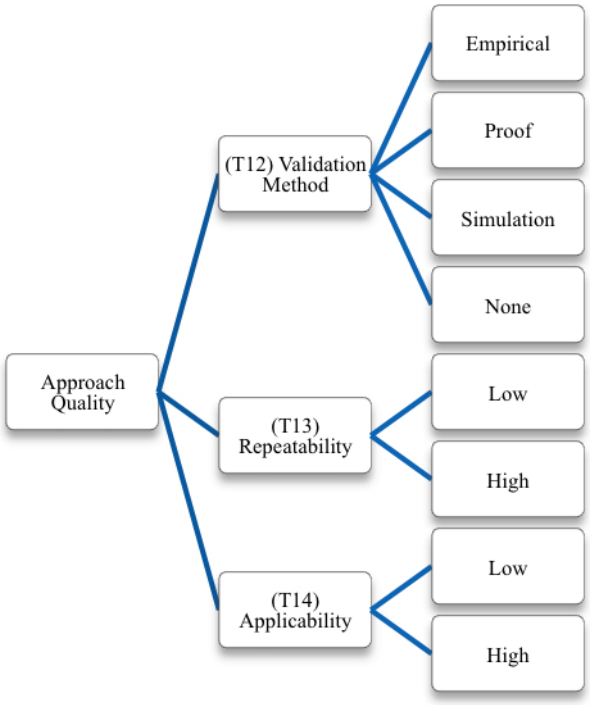


Figure 5.3: Proposed Taxonomy for Self-Protection Research (Cont.)

(T12) Validation Method

This dimension captures how a paper validates the effectiveness of its proposed approach, such as empirical experimentation, formal proof, computer simulation, or other methods. The selected sub-category for the validation method is closely related to the selected sub-category for the theoretical foundation (**T6**) of the proposed approach. When the approach is based on logic/formal methods, validation is expected to be in the form of formal proof. Approaches that are based on heuristics and optimization demand empirical validation. Finally, simulation is a perfect fit for learning based models.

(T13) Repeatability

This dimension captures how a third party may reproduce the validation results from a surveyed paper. This dimension classifies repeatability of research approaches using a simplified measure:

- *High repeatability*, when the approach's underlying platform, tools and/or case studies are publicly available;
- *Low repeatability*, otherwise.

(T14) Applicability

A self-protection approach or technique, though effective, may or may not be easily applied in a broader problem setting. Similar to repeatability, I use a simple "low" vs. "high" measure:

- *Low applicability*, when the approach is specific to a particular problem domain (suppose the online banking system uses users' income and spending patterns to calculate their risk profile), is dependent upon a proprietary framework or implementation, or requires extensive infrastructure support that may not be generally available;
- *High applicability*, otherwise.

5.4 Survey Results and Analysis

A large number of research efforts related to self-protecting systems and adaptive security have been identified in this survey, and are then evaluated against the proposed taxonomy. The detailed evaluation results are included in Appendix A.

Note that the classifications are meant to indicate the primary focus of a research paper. For example, if a certain approach does not have a checkmark in the "Availability" column under Protection Goals, it does not necessarily indicate that it absolutely cannot help address availability issues. Rather, it simply means availability is not its primary focus.

By using the proposed taxonomy as a consistent point of reference, many insightful observations surface from the survey results. The number of the research papers surveyed will not allow elaboration on each one of them in this thesis. Rather, I highlight some of them as examples in the observations and analysis below.

5.4.1 Correlating Self-Protection Levels and Depths of Defense

Starting with the Self-Protection Levels (**T1**) dimension, we see that abundant research approaches focus on the *Monitor & Detect* level, such as detecting security-relevant events and enforcing security policies that respond to these events. For example, Spanoudakis et al. [163] used Event Calculus to specify security monitoring patterns for detecting breaches in confidentiality, integrity and availability. Liang and Sekar [107] used forensic analysis of victim server's memory to generate attack message signatures. At the *Respond & Protect* level, research efforts attempt to characterize and understand the nature of security events and select the appropriate countermeasures. For example, He et al. [75] used policy-aware OS kernels that can dynamically change device protection levels. Taddeo and Ferrante [171] used a multi-attribute utility function to rank the suitability of cryptographic algorithms with respect to the runtime environment and then used a knapsack problem solver to select optimal algorithm based on resource constraints. At the highest *Plan & Prevent* level, research efforts are relatively speaking not as abundant; such efforts seek to tackle the harder problem of planning for security adaptation to counter existing and future threats. To that end, many approaches offer a higher degree of autonomicity. Uribe and Cheung [178], for instance, used a formal network description language as the basis for modeling, reasoning, and auto-generating Network-based Intrusion Detection System (NIDS) configurations. The Self-Architecting Software SYstems (SASSY) framework, by contrast, achieves architecture regeneration through the use of Quality of Service scenarios and service activity schemas [111, 118].

Along the Depths-of-Defense Layers (**T2**) dimension, we see many self-adaptive security

approaches focusing on the traditional architecture layers, such as network, host, middleware and application code. At the network level, abundant research can be found in the field of intrusion-detection and intrusion-prevention. Examples include Yu et al. [200, 201] who used fuzzy reasoning for predicting network intrusions, and the Wireless Self-Protection System (WSPS) [57] which uses both standard and training based anomaly behavior analysis that can detect and deter wide range of network attack types. Because network vulnerabilities are closely linked to the network topology and equipment configurations, devoted research can also be found on adapting network security policies based on such network characteristics [23]. At the host/node level, antivirus and malware detection/prevention have been receiving a lot of attention from the research community (a latest example on adaptive rule-based malware detection can be found in [20]).

When we shift our focus to defense at the middleware level, self-protection approaches start to focus and/or leverage distributed middleware platforms such as Java Enterprise Edition or JEE (as in [47]), object request brokers (as in [198]), and message-oriented middleware (as in [2] and [1]). More importantly, researchers started to recognize the benefit of a robust middleware layer as an extra line of defense against host and application level attacks (as seen in the QuO adaptive middleware example [10, 11]). More recent research has started to focus on adaptive security for web services middleware in a SOA. Such research can be found, for example, around service trust [116] and service-level assurance [29]. Research around the security behavior of a collection of services (such as a BPEL orchestration or a composite service), however, seems to be lacking.

As we move up to the application level, self-adaptive security research is more concerned with programming language level vulnerabilities such as those concerning pointers, memory buffers, and program execution points. Lorenzoli et al. [110], for example, presented a technique, called From Failures to Vaccine (FFTV), which detects faults using code-level assertions and then analyzes the application to identify relevant programming points that can mitigate the failures.

Research seems to be sparse on the adaptation of the software architecture as a whole

in dealing with security concerns. Nonetheless, the *Depth-Independent* subcategory in this dimension does capture some interesting and sophisticated approaches. The Rainbow [36,65] and SASSY frameworks are two examples that fit into this category, even though they are not specifically focused on self-protection alone. Additionally, work by Morin et al. [120] and Mouelhi et al. [122] represent a key example of applying "models@runtime" thinking to security adaptation, which can be applied to all architecture layers.

To take a further look at the research trends, we use Self-Protection Levels and Depths of Defense as two crosscutting dimensions to map out the existing self-protection research approaches, as shown in Figure 5.4. In the plot, the height of each column represents the number of papers per each self-protection level and each line of defense. We clearly see that abundant research exist at the network and host levels for attack detection and response, fueled by decades of research in such fields as Intrusion Detection / Intrusion Prevention (ID/IP), Antivirus / Malware, and Mobile Adhoc Networks (MANET) security. It becomes apparent, however, that existing research start to "thin out" as we move up the two respective dimensions. *Autonomic and adaptive security approaches that apply to attack prediction and prevention, especially at application or abstract architecture levels, appear to be a research gap to be filled.* This finding is one of the key motivations of our ABSP research.

5.4.2 Run-time vs. Development-time

Along the Lifecycle Focus (**T4**) dimension, the vast majority of self-protection research (96% to be exact) focuses on runtime. Indeed, it is a general consensus that software components are never completely fault-free and vulnerability-free no matter how carefully they are designed and coded. Nonetheless, 18% of the papers also involve development time activities. They generally fall under three cases:

- Runtime techniques that happen to need development time support. The FFTV approach [110], for instance, complements runtime healing/protection strategies with design-time construction of oracles and analysis of relevant program points, and also

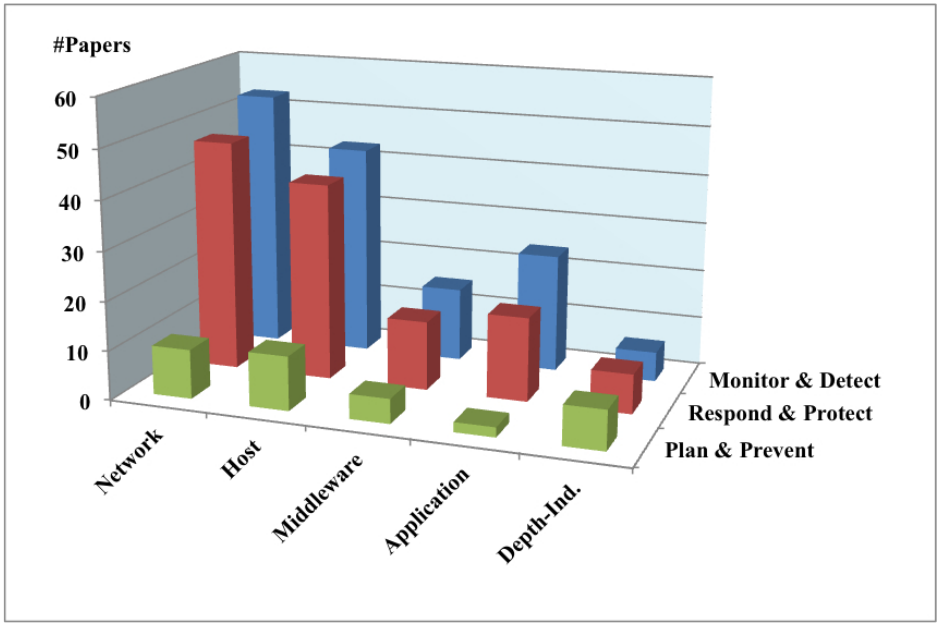


Figure 5.4: Correlating Self-Protection Levels and Depths of Defense

with test-time generation of reference data on successful executions. In [73], the dynamically reconfigurable security policies for mobile Java programs also rely on supporting mechanisms put in at deployment time (such as policy class loaders).

- Programming language level protection approaches that focus primarily at development time. They employ novel techniques such as fuzz testing [2], whitebox "data perturbation" techniques that involve static analysis [68,69], or software fault injection which merges security enforcement code with the target code at compile time [54].

- Model-driven approaches that essentially blur the line between development time and runtime. They achieve self-protection through incorporating security requirements into architecture and design models, and relying on Model-Driven Engineering (MDE) tool sets to instrument security related model changes at runtime. In addition to work by Morin et al. [120], the Agent-oriented Model-Driven Architecture (AMDA) effort [195, 196] also falls into this category. Such approaches may hold some promise

for future self-protection research, although empirical results so far are far from convincing.

Because the philosophy, structure, and process through which software components are constructed could have a significant impact on their quality of protection at runtime, I believe that full lifecycle approaches combining development-time and run-time techniques will result in the best self-protection of software systems another research opportunity.

5.4.3 Balancing the Protection Goals

Along the Protection Goals (**T3**) dimension, the survey results revealed that most research efforts seem to focus on either Confidentiality and Integrity or Availability, but not all three goals. As shown in the Venn diagram in Figure 5.5 (a), a large portion of the survey papers focus on Confidentiality (68%) and Integrity (81%), but only 40% of the papers address availability, and even fewer (20%) deal with all three goals. The dichotomy between confidentiality and availability objectives is not surprising: the former seeks mainly to protect the information within the system, but is not so much concerned with keeping the system always available; the opposite is true for the latter. For example, when a host-based intrusion is detected, the typical system responses involve stopping/restarting a service, rebooting the server, disable user logins, etc. [167] system confidentiality and integrity are preserved, whereas availability suffers.

Preserving system availability, on the other hand, goes beyond the security realm and is closely related to system QoS, thus requiring different treatments. Intrusion Tolerant Systems (e.g., [161] and [147]) address availability especially well by leveraging fault tolerance mechanisms, though they tend to focus on the network and host levels rather than taking a broader architectural approach.

This observation, though a bit subtle, shows that a self-protecting system may need to include a "best of breed" combination of adaptive security techniques rather than relying on a single mechanism, to meet all protection goals.

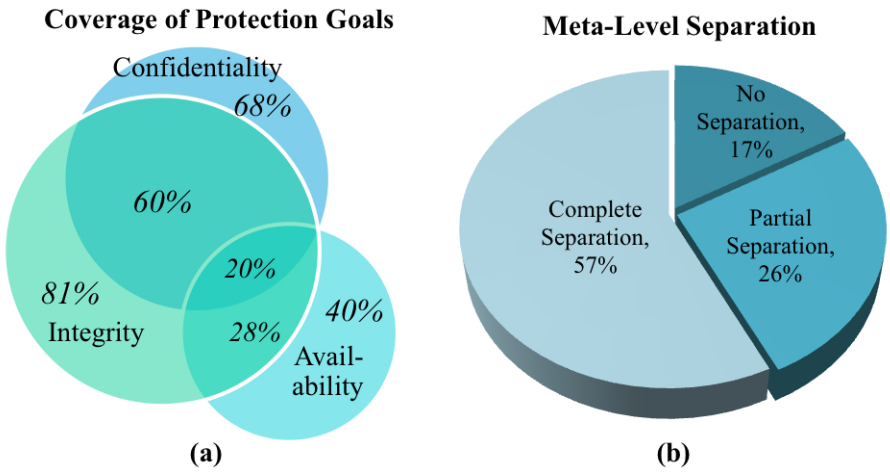


Figure 5.5: (a) Coverage of Protection Goals; (b) Meta-Level Separation

5.4.4 Separation of Meta-level and Base-level Subsystems

As introduced in Section 5.3.1, the Meta-level Separation dimension (**T5**) intends to show self-protection research separates the meta-level ("protecting") components from the base-level ("protected") components. The survey results summarized in Figure 5.5 (b) indicate that self-protection architectures from 83% of the papers show at least partial separation, which serves as strong evidence that the meta-level separation proposed in Chapter 4 has been indeed widely practiced in the research community. Instantiations of the meta-level subsystem take on many interesting forms among the surveyed papers, such as managerial nodes [1], Security Manager [16], guardians [119], Out-of-Band (OOB) server [149], or control centers [141]. Those approaches that have been put under "partial separation" either rely on certain enforcement mechanisms that are an intrinsic part of the base-level subsystem (e.g., through library interposition with the protected application [107]), or do not provide a clear architecture that depicts the separation boundary.

The remaining 17% papers that do not exhibit meta-level separation deserve special attention. Closer examination reveals two contributing factors are the primary "culprits", with the first having to do with the domain environment and the second pertaining to

the nature of the research technique. First, for MANETs, wireless sensor networks, or agent-based networks of a self-organizing nature, because no central control typically exists within the system, self-protecting mechanisms would have to be implemented within each network node or agent. This is the case with [3], [6], [39], and [86]. It is no coincidence that these papers also fall into the "Global Decentralized" subcategory of the Control Topology dimension (**T8**); see Section 5.4.6 for more details. Since each node/agent is just as susceptible to attack and subversion as any other node/agent, protecting the security mechanism itself becomes a real challenge. Costa et al. [44] used a mechanism called Self-Certifying Alerts (SCA) that are broadcasted over an overlay network to overcome this problem, but the challenge of "protecting the meta-level" in a globally decentralized topology is largely unanswered in the surveyed papers.

The second contributing factor arises from those approaches that use code generation and code injection techniques at the application level, because the protection mechanism becomes part of the codebase, meta-level separation is obviously lacking. It is quite revealing that most of the papers cited in Section 5.4.2 as having a development time focus such as FFTV [110], SASI [54], and AMDA [196] belong to this case! Here, we see another research challenge, that is, to find ways to employ valuable techniques (e.g., programming language analysis and model-driven engineering) while staying true to the self-protection reference architecture with clear meta-level vs. base-level separation.

5.4.5 Foundations and Strategies for Self-Protection Decision-Making

When it comes to the "HOW" part of the taxonomy, we see the surveyed papers employ a large variety of models, schemes, algorithms, and processes. First of all, a simple analysis along the Theoretical Foundation dimension (**T6**) shows a predominant use of heuristics-based methods, as shown in Figure 5.6 (a), in such forms as expert systems [125,139], policy specification languages [23], event-condition-action rules [53], directed graphs [14], genetic/evolutionary algorithms [145], structured decision analysis (such as Analytic Hierarchy Process or AHP, as in [16]), or human input as a last resort [192]. Even when non-heuristics

based methods are used, whether it is using formal semantics [50] or utility function based optimization [171] or stochastic modeling [162], they are more often than not complemented by heuristics. Our analysis along this dimension has revealed the following insights:

- Given the multitude of decision factors such as objectives, system properties, resource constraints and domain-specific environment characteristics, the problem space for self-protection decision making is usually too large for classic problem solving methods (though they may still prove effective in solving a small, narrowly-focused subproblem, such as formalisms and proofs around software rejuvenation [130] or reinforcement based learning for malware detection [20]);

- Because the entire system is at stake with regard to self-adaptive security decisions, a wrong move may lead to severe consequences. As such, few approaches in this survey leave such decisions (such as threat containment or deploying countermeasures) solely to an algorithm without any heuristic input. Indeed, as pointed out in a number of papers [5, 46], autonomic responses often require near 100% accuracy in threat detection and characterization (i.e. the rate of false positives at near zero). Many papers went to great lengths to analyse and reduce the rate of false positives while maintaining a high detection rate (i.e. low false negatives), but results vary.

- The lack of non-heuristics based methods may also be explained by the daunting challenge of quantitatively assessing the overall security posture of a complex software system. Several papers proposed various metrics as attempts to this goal the Security Health Index comprised of a weighted basket of security metrics [153] and the Compromise Confidence Index as a measure to pinpoint attack location [60] are representative examples. Empirical validation of these metrics, however, is far from sufficient and convincing from the surveyed papers. This is definitely a pressing research need, especially in today's heated domain of cyber warfare.

The meta-level decision making dimension (**T7**) of our taxonomy offers an even more interesting perspective on self-protection decision making. From Figure 5.6 (b) we can see

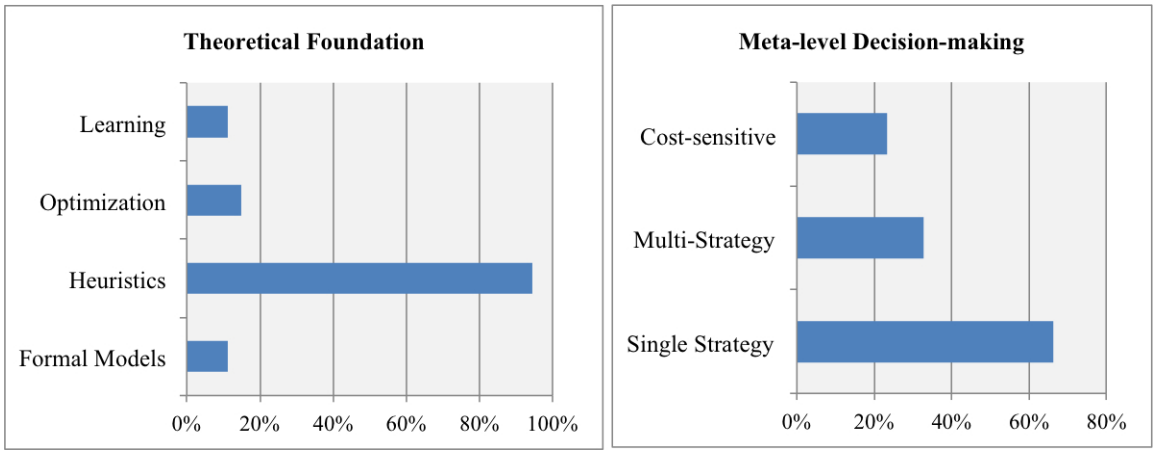


Figure 5.6: (a) Theoretical Foundation; (b) Meta-level Decision-making

two important opportunities in self-protection research:

From single-strategy to multi-strategy. Some researchers have come to the realization that a single technique or a point solution is no longer enough to match the ever increasing sophistication of today's cyber-attacks, as described in Section 3.1. Rather, self-protecting systems should be able to (1) detect higher-level attack patterns and sequences from low-level events, (2) have an arsenal of countermeasures and response mechanisms that can be selectively activated depending on the type of attack, and (3) have a concerted strategy to guide the selection and execution of the responses at multiple defense depths and resolve conflicts if necessary. A number of research papers have started down this path. The APOD initiative [10, 11], for example, uses higher level strategies (e.g. attack containment, continuous unpredictable changes, etc.) to derive/direct lower-level sub-strategies and local tactics in responding to attacks. Similarly, the AVPS approach [158, 159] generates signatures (low level rules) based on high level rules; Tang and Yu [173] showed that high-level goal management can optimize the lower level policy execution at the network security level. This is an encouraging trend, although the survey shows the multi-strategy based papers are still a minority (at 33%).

From security-at-any-cost to cost-sensitive protection. Though earlier attempts exist in quantifying the cost of intrusion detection and prevention (such as [100]), an increasing number of recent research papers start to consciously balance the cost (that is, penalization of other quality attributes) and benefits of autonomic responses to security attacks. Stakhanova et al. [165,167], for example, defined a set of cost metrics and performed quantitative trade-off analyses between cost of response and cost of damage, and between the importance of early pre-emptive responses (when there is a high cost of missed or late detections) vs. the importance of detection accuracy (when there is a high cost of false positives). Similarly, Nagarajan et al. [124] developed cost models involving operational costs, damage costs, and response costs, and implemented the model using Receiver Operating Characteristic (ROC) curves. At 23%, the cost-sensitive strategies are a minority but I believe they represent a promising and sensible direction for self-protection research, especially in the larger picture of self-* systems.

Both opportunities call for a holistic approach to self-protection, yet another driver for the ABSP approach: with the global view of the system afforded by the archtiecture representation including the interdependencies among the system's constituents, the ABSP approach can help determine the impact of a compromised element on the other parts of the system and formulate globally coordinated defense strategies.

5.4.6 Spatial and Temporal Characteristics

Together, the three taxonomy dimensions Control Topology (**T8**), Response Timing (**T9**), and Enforcement Locale (**T10**) expose interesting characteristics and trends about the spatial and temporal aspects of self-protection approaches that is, where the "brain" of the self-protection is within the system and where the "action" takes place, as well as when the adaptive actions are carried out.

First, as shown in Figure 5.7, survey results along the Control Topology dimension clearly shows that adaptive security approaches functioning at the global level are predominantly *centralized* about 57% of the papers. For example, many research efforts (e.g., [75]

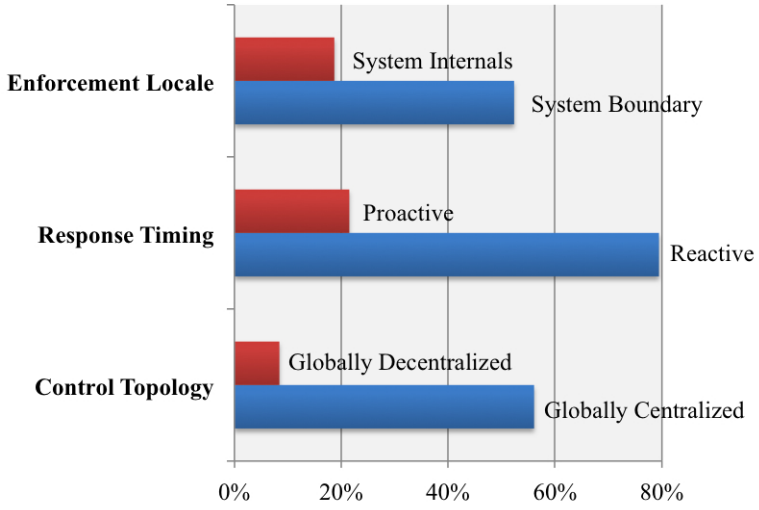


Figure 5.7: Temporal and Spatial Characteristics

and [2]) recognize the need for coordination between local and global security policies. In most cases, the coordination is through a central controller (as used in [79]). One of the main reasons behind widespread adoption of centralized topology may be the fact that using a central controller makes coordination and global optimization easier. However, central controller runs the risk of becoming the single point of failure of the system, prone to denial of service and subversion attacks. Some approaches put more robust protection around the central controller, such as using hardened and trusted hardware/software (as in the case of the Malicious-and Accidental-Fault Tolerance for Internet Applications (MAFTIA) system [182]) or putting the controller in dedicated network zones [40]. Another potential disadvantage for the centralized approach is scalability. For pervasive systems with highly distributed computing resources, it may be inefficient and costly to have all of the resources communicate with a central controller. Accounting for only 8% of the total papers, globally *decentralized* approaches appear to be an exception rather than norm. As pointed out in section 5.4.4, self-protection efforts from the MANET and self-organizing agent domains tend to fall into this category because the system topology does not allow

for a centralized component. The decentralized control topology is not limited to these domain environments however. MAFTIA, for example, also uses local middleware controllers (called "wormholes") at each server that are interconnected yet do not appear to require a central controller. Decentralized security approaches hold more promise in their resilience and scalability. The fact that coordination and global optimization is harder in a decentralized setting indicates the need for more research attention. Indeed, decentralized control has been highlighted as a key research topic on the roadmap of self-adaptive systems [104].

Secondly, survey results along the Response Timing dimension indicate *reactive* adaptation based on the "sense and respond" paradigm still seems to be the norm for self-protection (79% of total papers). That being said, the survey results also show an interesting trend that proactive security architectures are gaining ground in the past decade, with 21% papers claiming some proactive tactics. By proactively "reviving" the system to its "known good" state, one can limit the damage of undetected attacks, though with a cost. The TALENT system [128, 129], for example, addresses software security and survivability using a "cyber moving target" approach, which proactively migrates running applications across different platforms on a periodic basis while preserving application state. The Self-Cleansing Intrusion Tolerance (SCIT) architecture [124] uses redundant and diverse servers to periodically "self-cleanse" the system to pristine state. The aforementioned R-Xen framework [85] proactively instantiates new VM instances to ensure system reliability, a technique much faster than rebooting hardware servers thanks to hypervisor-based virtualization technology.

Thirdly, the Enforcement Locale dimension shows that over 52% of self-protection approaches still rely on perimeter security, especially those that focus on intrusion detection and intrusion prevention. Systems relying solely on perimeter security, however, are often rendered helpless when the perimeter is breached; nor can they effectively deal with threats that originate from inside of the system. To compensate for this weakness, some approaches follow the "defense-in-depth" principle and establish multiple layers of perimeters or security zones [134], but the disadvantage still exists. In light of this, I feel there is a need to shift focus from perimeter security to overall system protection, especially from monitoring

the system boundary to monitoring overall system behavior. For example, recent research has started to focus on detecting and responding to insider threats based on monitoring user-system interactions [158, 159]. Another possible approach is to shift the focus from delimiting system boundaries to identifying system assets under protection, as developed by Salehie et al. [151] and Pasquale et al. [136].

Figure 5.7 summarizes the statistics around the spatial and temporal traits of surveyed approaches, highlighting the research gaps around (1) global self-protection architectures that do not require a central controller, (2) combining reactive protection tactics with proactive ones, and (3) protecting the overall system and not just the perimeter.

5.4.7 Repeatable Patterns and Tactics for Self-Protection

One of the most revealing findings from our survey is the emergence of repeatable architectural patterns and design tactics that software systems employ specifically for self-protection purposes (**T11** of the taxonomy). Even though some of these patterns bear similarity to the generic software architecture and design patterns or security patterns [70, 199], their usage and semantics are quite different. As mentioned in Section 5.3.2, they can be loosely categorized as structural and behavioral patterns. The major structural patterns I identified include:

- Protective Wrapper
- Agreement-based Redundancy
- Implementation Diversity
- Countermeasure Broker
- Aspect Orientation

And the behavioral patterns include:

- Protective Wrapper

- Agreement-based Redundancy
- Implementation Diversity
- Countermeasure Broker
- Aspect Orientation

I will defer their detailed description, examples, and perceived pros/cons until Chapter 9. Suffice to say here that these patterns cover 84% of the surveyed papers; therefore their use is quite pervasive. Note that the patterns are by no means mutually exclusive.

I also found, not surprisingly, that the positioning and techniques employed by a self-protection approach will to some extent determine the architectural patterns being used. This observation, however, does point to a critical research opportunity, that is, to further identify and catalogue such correlations, to codify them into machine-readable forms, so that a system may dynamically re-architect itself using repeatable patterns as requirements and environments change. This is a higher level of self-protection and may only be enabled through an architecture-based approach.

5.4.8 Quality Assessment of Surveyed Papers

I used reputable sites in our review protocol (recall Section 5.2). This resulted in the discovery of high quality refereed research papers from respectable venues. I use Validation Method (**T12**), Repeatability (**T13**), and Applicability (**T14**), which are the three Approach Quality dimensions in the taxonomy, to develop better insights into the quality of the research papers surveyed. Figure 5.8 summarizes some of our findings. Figure 5.8 (a) depicts the share of different Validation Methods in assessing the quality of self-protection approaches. Most (70%) of the approaches have used Empirical techniques to assess the validity of their ideas. The empirical techniques range from detailed assessment of full implementation of the approach (e.g., [30]) to a proof-of-concept experimentation of a prototype (e.g., [145]). A limited number (6%) of approaches (e.g., by Ostrovsky and Yung [130]) have provided mathematical Proof to validate their ideas. Some approaches (16%) have relied

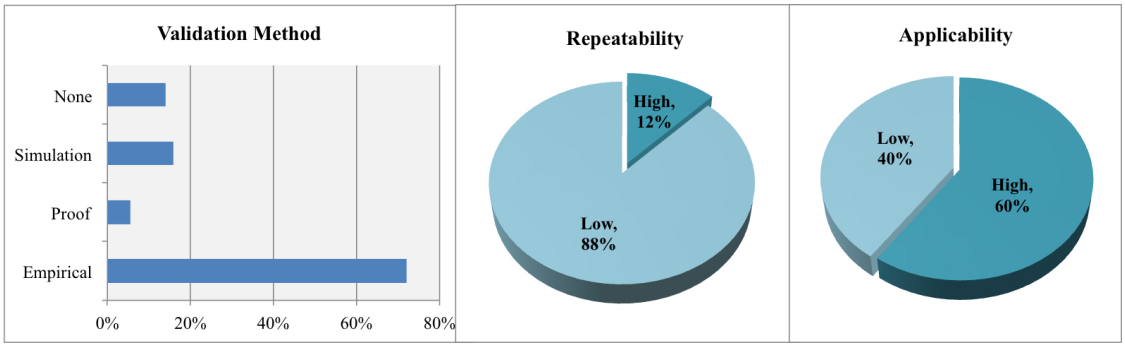


Figure 5.8: (a) Validation Method (b) Repeatability (c) Applicability

on simulation to validate their claims (e.g., [171]). Finally, there are approaches (14%) that have either not validated their ideas (e.g., [179]) or validated aspects (such as performance) of their work other than security (e.g., [170]).

The evaluation of security research is generally known to be difficult. Making the results of experiments repeatable is even more difficult. We can see this in Figure 5.8 (b), where only limited portions (12%) of approaches are highly repeatable. The rest have not made their implementations, prototypes, tools, and experiments available to other researchers. This has hampered their adoption in other relevant domains.

As we can see in Figure 5.8 (c), many approaches (60%) are portable and have high potential for being applicable to broad range of situations and domains. However, the fact that their artifacts are not accessible outside the boundary of the same team/organization, has limited their usage and prevented their potential applicability from becoming actual applicability.

5.5 Recommendations for Future Research

From systematically reviewing the self-protection related literature, we see some important trends in software security research. Starting in the 1990s, dynamic and automated security mechanisms started to emerge in the antivirus and anti-spam communities. The late 1990s

and early 2000s saw a research boom in the Intrusion Detection (ID) / firewall communities, as online systems faced the onslaught of network and host based attacks. We then see two important trends in the past decade, as reflected by the observations in section 5.4: (a) *from Intrusion Detection to Intrusion Response (IR)*, as the increasing scale and speed of attacks showed the acute need for dynamic and autonomic response mechanisms, as confirmed in recent surveys [166] and [157]; (b) *from Intrusion Detection to Intrusion Tolerance (IT)*, when both the industry and academia came to the realization that security threats will persist and prevention mechanisms will likely never be adequate, and began to give additional consideration to maintaining system performance even in the presence of an intrusion [126].

Only in recent few years did we see an explicit focus on self-protection as a system property in the autonomic computing context. Frincke et al. [62] and Atighetchi and Pal [9], for example, see the pressing need for autonomic security approaches and true self-protecting systems. From the break-down of papers across ID, IR, IT, and Self-Protection (SP) communities, we see an encouraging sign of growing SP research, yet we also see the continued influence of the intrusion-centric mindset.

Therefore, our first and foremost recommendation is to **increase attention, convergence, and collaboration on self-protection research**, and to leverage this community for integrating a diverse set of strategies, technologies and techniques from ID, IR, IT and other communities toward achieving a common goal.

More specifically, the survey using our proposed taxonomy has revealed some gaps and needs for future research. To summarize, self-protection research needs to focus on the following to stay ahead of todays advancing cyber threats:

- Advance the sophistication at each self-protection level, that is, from automatically monitoring and detecting threats and vulnerabilities to autonomously predict and prevent attacks;

- Move beyond defending the network and host layers, towards developing approaches

that are independent of specific system architectures and that can select suitable strategies and tactics at different architecture layers;

- Pursue integrated, "full lifecycle" approaches that span both development-time and runtime;
- "Protect the protector", that is, safeguard the meta-level self-protection module, especially in a globally decentralized topology;
- Explore models@runtime and model-driven engineering techniques while maintaining clear meta-level and base-level separation;
- Explore the definition and application of qualitative and quantitative metrics that can be used to dynamically assess overall system security posture and make autonomic response decisions;
- Continue to explore multi-level, multi-objective, as well as cost-sensitive security decision-making strategies based on stakeholder requirements;
- Continue the paradigm shift from perimeter security to overall system protection and monitoring, and from merely reactive responses to a combined use of both reactive and proactive mechanisms;
- Catalog, implement, and evaluate self-protection patterns at the abstract architecture level;
- Promote collaboration in the community by making research repeatable (such as providing tools and case studies) and developing common evaluation platforms/benchmarks.

Chapter 6: Discovering the Architecture Model Through Machine Learning

In this chapter, I start delving into the details of the ABSP approach but focus specifically on the development of an architecture model at runtime with the aim of validating Hypothesis #1 introduced in Section 3.2. I start with introducing a real world system called EDS which is used as a system of reference throughout my research. I then proceed to introduce two closely related mining techniques that mines the component interaction events captured in system execution history: the Association Rules Mining (ARM) algorithm, which produces a model based on the usage proximity of the software components, and the Generalized Sequential Pattern (GSP) Mining technique, which further takes advantage of the temporal information inherent in the execution history. The resulting models will then be used in the next two chapters for detecting anomalous behavior.

6.1 Reference System Description

I illustrate the concepts and evaluate the research using a software system, called Emergency Deployment System (EDS), which is intended for the deployment and management of personnel in emergency response scenarios. Figure 6.1 depicts a subset of EDS's software architecture, and in particular shows the dependency relationships among its components. EDS is used to accomplish four main tasks: (1) track the emergency operation resources (e.g., ambulances, rescue teams, etc.) using Resource Monitor, (2) distribute resources to the rescue teams using Resource Manager, (3) analyze different deployment strategies using Strategy Analyzer, and finally (4) find the required steps toward a selected strategy using Deployment Advisor. EDS is representative of a large component-based software system, where the components communicate by exchanging messages (events). In the largest

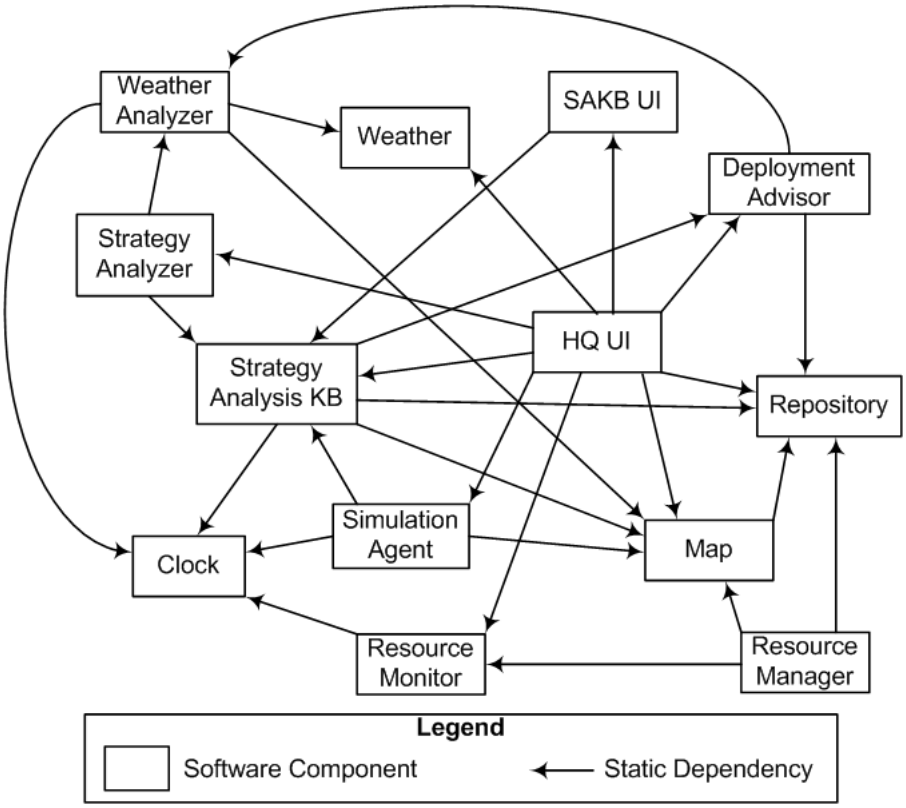


Figure 6.1: Subset of EDS Software Architecture

deployment of EDS to-date, it was deployed on 105 nodes and used by more than 100 users [113].

EDS is representative of a large *component-based* software system, with each component representing a coarsely grained software unit that deploys and runs independently from other components [45], possibly at different host nodes and locations (in contrast to lower level entities such as a Java object or a code library). For instance, a component could be an Enterprise JavaBean or a web application that resides on a web server. For EDS, in particular, each component is a Java Remote Method Invocation (RMI) application, discoverable from a registry and invoked over the network by any RMI client application.

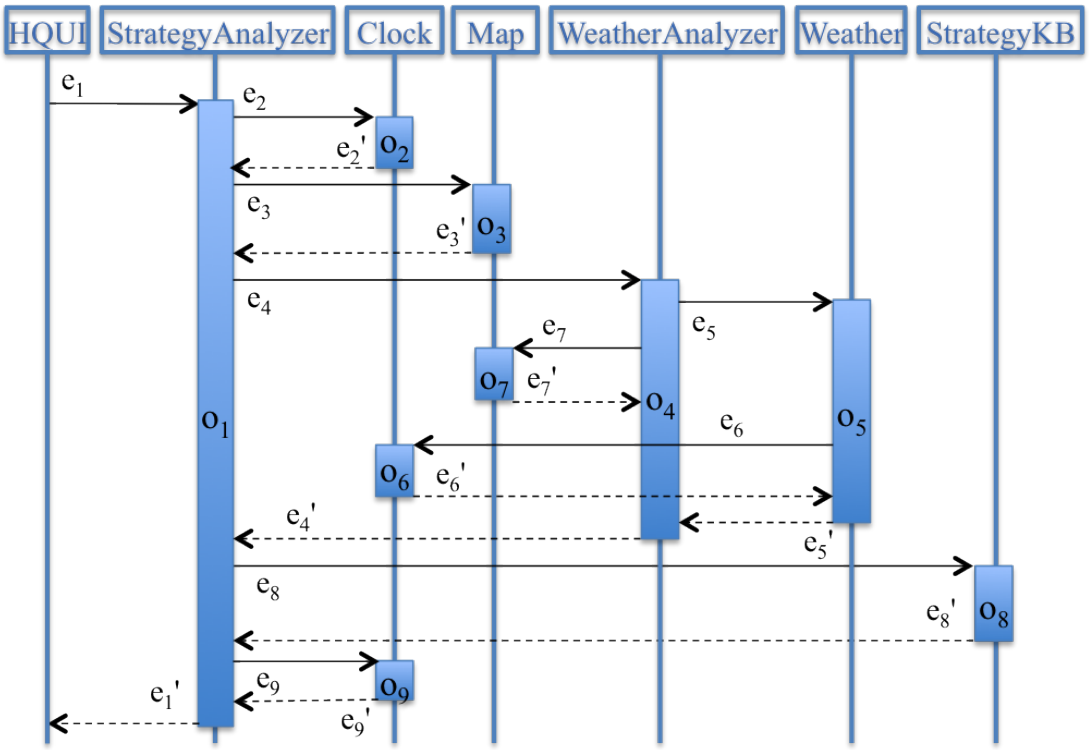
Like any software system, the EDS functionality can be decomposed into a number of user requirements or *use cases*. The sequence diagram for two such use cases, conducting

strategy analysis and conducting scenario simulation, are shown in Figure 6.2 as examples. We see that the execution of a use case involves a sequence of interactions among different software components. Note that the components interactions may utilize various communication mechanisms, from shared memory access for collocated components on the same host to publish-subscribe messages across a Wide Area Network. My architecture model, however, is only concerned with the logical relationship and is thus independent of the networking properties.

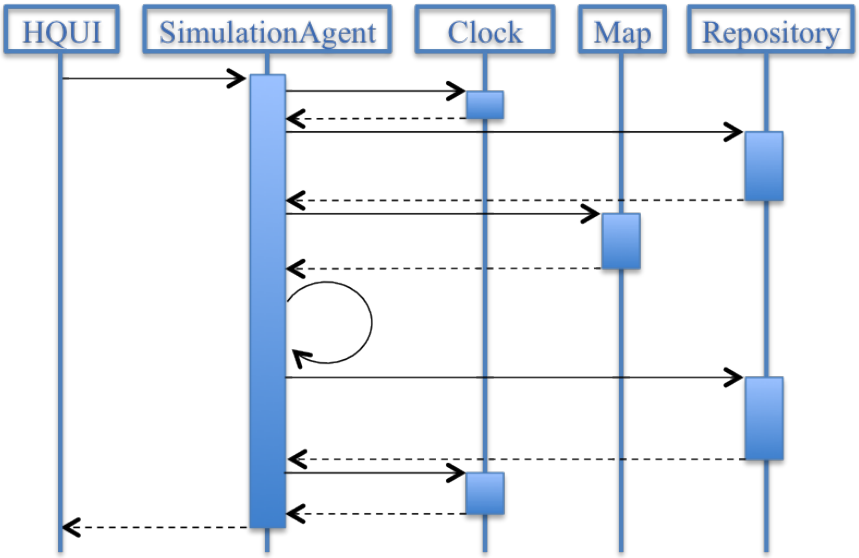
Once deployed and operational, a real-world system such as EDS needs to continually evolve to ensure quality, meet changing user requirements, and accommodate environment changes (such as hardware upgrades). The system must satisfy a number of architectural objectives such as availability, performance, reliability, and security.

Non-trivial system adaptations, such as those for self-protection purposes, typically require an abstract representation of the components and their interactions at runtime, which can be used to formulate adaptation strategies and tactics [65]. In the case of EDS, a model such as Figure 6.2 could be used for a variety of purposes. For instance, as shown in [180], it could be used to determine Weather Analyzer can be safely adapted (e.g. for version upgrades) prior to event e_4 or after event e'_4, but not in between, as its state is inconsistent.

Building and maintaining such a component interaction model, however, faces several difficult challenges. First, in a complex software system, manually defining models that represent the component interactions is time consuming. Second, it is not always possible to construct such models *a priori*, before the system's deployment. In service-oriented architectures (SOA) or peer-to-peer environments, for instance, component behavior may be user-driven and non-deterministic. Third, even when such models are built, it is a heavy burden to keep them in sync with the actual implementation of the software. Indeed, they are susceptible to the well-studied problem of architectural decay [174], which tends to occur when changes applied to the software are not reflected in its architecture models.



(a) UC1 — Strategy Analysis Use Case



(b) UC2 — Scenario Simulation Use Case

Figure 6.2: Examples of EDS Component Interactions

My approach to addressing these challenges involves learning a *usage proximity* model of dynamic component interactions at runtime, without any pre-defined behavior specifications. Machine learning-based approaches alleviates engineers from maintaining the models manually, and also allow for their automatic adaptation and refinement to changing behavior of the system and its environment.

6.2 Definitions and Assumptions

I start with some basic definitions and assumptions to frame the discussion of the proposed approach.

First, let $C = \{c_i\}$ be a set of **software components** that run independently and are capable of sending and receiving messages among one another.[1] An **event** is defined as a message from a source component to a destination component, captured as a tuple $e =< c_{src}, c_{dst}, t_s, t_e >$, where $c_{src}, c_{dst} \in C$ and t_s, t_e are the start and end timestamps of the event's occurrence, respectively. In Figure 6.2 (a), for example, event e_2 is a message from StrategyAnalyzer to the Clock component, also denoted as $StrategyAnalyzer \rightharpoonup Clock$.

Here I assume the network clock is synchronized (e.g., using the Network Time Protocol [177]) such that timestamps for all system events are properly recorded within a margin of error, $\varepsilon_{NTP} \ll t_e - t_s$; This is a fairly safe assumption for EDS since the inter-component messages are at least hundreds of milliseconds or longer apart while the ε_{NTP} is usually in the milliseconds range, but nonetheless I will account for this margin in the mining process in Section 8.2.1.

An **operation** O_i is performed in a component when receiving a message e_i. Note that even though Figure 6.2 depicts events that happen to be *synchronous*, i.e., a response message goes back to the source component at the end of O_i (as in the case with many request-response protocols like HTTP or Java RMI), I do NOT rely on such an assumption; an event may also be asynchronous, without waiting for the completion of the operation.

[1] For simplicity we assume each component has a single interface or "port". In reality different types of messages may be going through different ports, in which case we can simply add ports to the tuple

In the latter case, the duration of the event is simply the network latency.

An event's **Perceived Execution Closure** (PEC) e^+ is an itemset that includes the event itself plus all child events that *may* have been triggered by the event. In Figure 6.2 (a), for example, events e_5, e_6, and e_7 are all triggered by e_4, therefore $e_4^+ = \{e_4, e_5, e_6, e_7\}$. Here the closure is considered *perceived* because I do not assume we know the true causality among events; they can only be inferred by the starting and ending components and the timestamps of events.

A **Use-case Initiating Event** (UIE) represents a "top level" event that is not part of the execution closure of any other events. For EDS, a UIE naturally corresponds to an "entry point" at one of the system's user interfaces such as simulation analysis or strategy analysis. In Figure 6.2 (a), for instance, e_1 is the UIE that initiates events e_2 through e_9.

To keep our approach widely applicable, I make minimal assumptions about the available information from the underlying system:

- *Black-Box Treatment*: Even though the components of the system are known (as depicted in Figure 6.1), I assume the software components' implementation or source code is not available, which is normal for services or COTS components. Further, I assume the *dynamic* behavior model of the target system, like what is shown in Figure 6.2, is not specified or is even non-deterministic due to reasons cited in the previous section.

- *Observability of Events*: I assume that events marking the interactions among the system's components are observable. An event could be either a message exchange or a method call, which could be monitored via the middleware facilities that host the components or instrumentation of the communication links. Web-based systems, for instance, typically have web server logs that can be filtered and processed for this purpose. In EDS, we developed a common logging service that stores all RMI calls in a database.

- *UIEs, that is, top-level user transactions, can be identified.* Here I assume that a

number of "entry point" events exist that initiates top-level transactions. Such events typically represent the starting point of a system use case. An online banking system, for example, may have menu items such as "Withdrawal", "Deposit", or "Check Balance" that trigger different use cases. The EDS system, likewise, has client-server events (such as e_1 in Figure 6.2 (a)) that initiate different use cases.

6.3 Association Rules Mining (ARM)

Association Rules Mining (ARM) [172], also known as frequent pattern mining, is a useful technique for discovering hidden relations in large datasets. The mining algorithms takes as input a large number of *itemsets*, and examines the co-occurrence of individual items within the itemset. Using a retail example, a customers' purchase transaction at a store may be viewed as such an itemset, with each transaction containing one or more purchased items. ARM techniques have been historically used to answer questions such as "If a customer buys diapers, what other products is he/she also likely to buy?" Here in our problem setting, the items are component interaction events from the system's execution history, and given the observability of the UIEs, an itemset is naturally the "basket" of events that fall into the same UIE or top level user transaction. Applying the mining algorithm over the event itemsets generates rules that reveal how closely one event may co-occur with other events at runtime.

Several association mining algorithms exist such as Apriori [4] and FP-Growth [72]. I primarily used the Apriori algorithm due to the fact that its implementations are mature and widely available. The algorithm produces **Event Association Rules (EAR)** of the form

$$X \Rightarrow Y : < supp, conf > \tag{6.1}$$

where X and Y are sets of events, e.g. $X = \{e_i, e_j\}$, $Y = \{e_k\}$ and $supp$ and $conf$ are the

support and *confidence* level of the rule, respectively:

$$supp = \sigma(X \cup Y)/N \tag{6.2}$$

$$conf = \sigma(X \cup Y)/\sigma(X) \tag{6.3}$$

where $\sigma(S)$ is the count of all itemsets to which S is a subset (that is, frequency of co-occurrence of events in set S), and N is the total number of itemsets. Support level *supp* indicates how frequent a rule is applicable and is typically used to eliminate "uninteresting" rules, while confidence $conf$ measures the reliability of the inference made by the rule.

The Apriori algorithm takes two main parameters as input, the minimum support level $minsupp$ and the minimum confidence $minconf$. During an interative process, candidate rules are generated with increasing length (that is, size of $X \cup Y$), and those rules with support below $minsupp$ or confidence below $minconf$ are eliminated.

As a concrete example, here is a rule generated in one of the test runs for the EDS system:

$$\{HQUI \rightharpoonup StrategyAnalyzer,\ StrategyAnalyzer \rightharpoonup Clock,$$
$$StrategyAnalyzer \rightharpoonup StrategyKB\} \Rightarrow \{StrategyAnalyzer \rightharpoonup Map\} : < 0.25, 0.65 >$$

which tells us that the four events appear together in 25% of all user transactions in the training data set, and whenever the three events on the left-hand side co-occur, there is a 65% chance that event $StrategyAnalyzer \rightharpoonup Map$ on the righ-hand side is also present in the same user transaction.

It is important to note that an EAR of the form $\{e_i, e_j\} \Rightarrow \{e_k\}$ does <u>not</u> represent a temporal execution sequence, since both the left-hand side and the right-hand side are (unordered) sets. Rather, as an association rule it simply states the frequency of event e_k occurring together with events e_i and e_j in one user transaction.

Here we can easily see that the resulting set of EARs represent a *usage proximity model* for the normal system behavior at runtime, answering my Hypothesis #1 from section 3.2.

The higher confidence of the EAR rule, the more likely its constituent components interact with one another in a user transaction. As an example of the utility of this model, Canavera et al. [24] was able to use the model to infer the dependencies among the system's components to ensure their adaptation (e.g., replacement) does not leave the system in an inconsistent state, with more flexibility than previously proposed models such as *Quiescence* [92] and *tranquility* [180].

6.4 Generalized Sequential Pattern (GSP) Mining

Events from the system execution traces are naturally sequenced by their timestamps, but the temporal information is not captured in the EARs from associations mining, which only captures co-occurrence of the events. How do we capture the temporal information in the architecture model? This where sequential pattern mining comes in.

Sequential pattern mining [164], also known as frequent episode mining, is a technique for discovering frequently occurring patterns in data sequences, used in a wide range of application domains from retail industry to biological research. In the context of sequential pattern mining, a *sequence* is defined as an ordered list of itemsets, where each itemset is a set of literals called items. We denote a sequence by $\prec s_1, s_2, ..., s_n \succ$, where s_i is an (unordered) itemset, denoted by $(x_1\ x_2\ ...\ x_k)$. Also using a retail analogy, a customer's purchase history at a store may be viewed as a data sequence consisting of a list of transactions ordered by transaction time, with each transaction s_i containing one or more purchased items.

A sequence $\prec a_1, a_2, ..., a_n \succ$ is a *subsequence* of another sequence $\prec b_1, b_2, ..., b_m \succ$ if there exists integers $i_1 < i_2 < ... < i_n$ such that $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, ..., a_n \subseteq b_{i_n}$. For example, $\prec c, (d\ e) \succ$ is a subsequence of $\prec g, (c\ h), k, (d\ e\ f) \succ$, but $\prec d, e \succ$ is not a subsequence $\prec (d\ e) \succ$ (and vice versa). Given a set of data sequences, the objective of sequencial pattern mining is therefore to find all subsequences that meet a user-defined minimum frequency (also called *support*). Continuing with the retail example, mining customer transactions

may reveal a pattern like "15% customers bought Star Wars movies, followed by Lego Star Wars toys in a later transaction".

A variety of sequential pattern mining algorithms exist; I chose to use the well-known GSP algorithm [164], due in part to its available open-source implementation [71]. I use GSP to discover **Event Association Patterns (EAP)** of the form:

$$P = \prec s_1, s_2, ..., s_n \succ : \ supp \tag{6.4}$$

where each element s_i is an itemset of co-occurring events and support $supp$ is the count of all sequences to which P is a subsequence, divided by the total number of data sequences: $supp = \sigma(P)/N$. Naturally $supp \in [0, 1]$, and the more frequently P occurs, the higher the $supp$. As a concrete example, here is an EAP generated from GSP test runs:

$$\prec HQUI \rightharpoonup StrategyAnalyzer,\ (StrategyAnalyzer \rightharpoonup Clock$$
$$StrategyAnalyzer \rightharpoonup StrategyKB) \succ : \ 0.45$$

The original GSP algorithm follows an iterative process that generates candidate sequences with $(k + 1)$ elements based on existing k-sequences, then prunes the candidates that do not meet the minimum support level; essentially the same strategy used in the Apriori algorithm for mining assocations among itemsets [4].

Similar to the EARs produced from the ARM technique, the set of EAPs produced from GSP mining also serve as a simple form of a system behavior model at runtime, answering my Hypothesis #1 from section 3.2. Much like the shotgun genome sequencing method that reads a large number of DNA fragments to provide a consensus reconstruction of a contiguous genome [181], the EAPs collectively provide insight into how the components interact with one another at the architecture level, with added temporal knowledge.

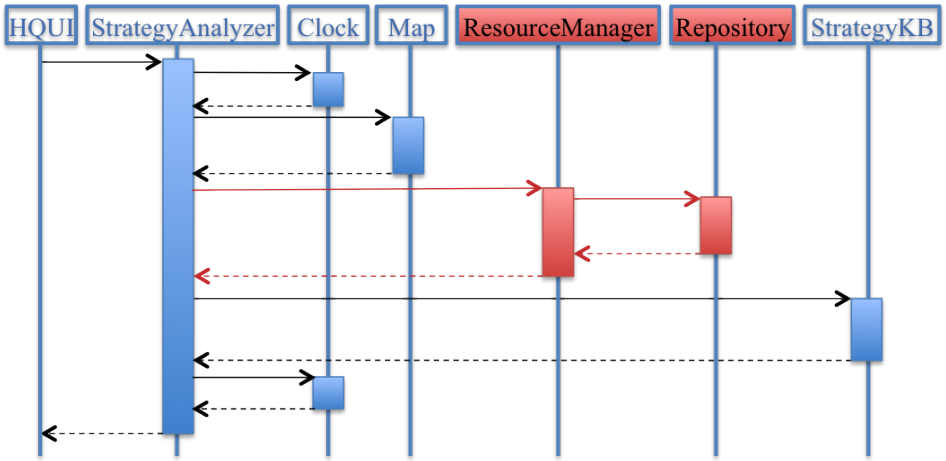
In the next two chapters, respectively, I will apply the ARM and GSP models in an unconventional way for detecting anomalous and potentially malicious interactions among the system components. By evaluating both models in comparison, we can see some comparative advantages of the GSP algorithm over ARM.

Chapter 7: Architectural-Level Security Anomaly Detection: the ARMOUR Framework

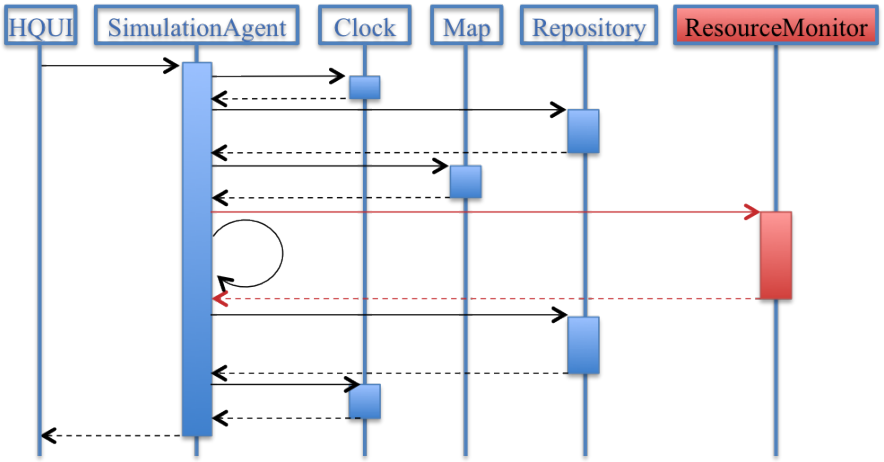
I now proceed to define the centerpiece of the ABSP approach, an adaptive, use case-driven framework for applying a form of architecture model, namely, the component interaction model introduced in the previous chapter, to help detect anomalous and potentially malicious behavior that may otherwise slip through conventional security mechanisms. In this chapter I describe the initial implementation and evaluation of the ARMOUR framework based on a tailored ARM algorithm, and in the next chapter I show how the framework can be enhanced by the GSP algorithm which is better-performing and more efficient.

7.1 Threat Analysis

In today's environment of ubiquitous connectivity, online systems such as EDS are often subject to various exploits and attacks, both external and internal. Intrusion detection sensors (IDS) have been developed and perfected over the years to effectively detect network-level (such as port scanning and denial of service) and host-level (such as buffer overflow or illegal file access) attacks. Attacks at the *software application* level, however, are often more sophisticated and much harder to detect, especially when they exploit vulnerabilities in seemingly innocuous user interfaces such as a web browser or a mobile app. Consider a scenario in which an attacker employs SQL Injection [176] through the browser-based Headquarters User Interface (HQUI) component and successfully compromises the *StrategyAnalyzer* component, which is a Java Servlet application residing in a web server. Using a malicious database script to tamper the component's configurations, the attacker is able to make *StrategyAnalyzer* send requests to a *ResourceManager* component to retrieve sensitive information about all deployed resources, as shown in Figure 7.1 (a) (Compare with



(a) Compromised UC1 with SQL Injection Attack



(b) Compromised UC2 resulting from Insider Threat

Figure 7.1: Examples of EDS Attack Cases

Figure 6.2 (a)). SQL Injection attacks are considered a top threat to web applications today [133].

As another example, the HQUI client may be used by a rogue employee to log into *SimulationAgent* using valid credentials. Like many real systems, EDS does not strictly enforce least-privilege access, therefore the employee is able to rewrite URL parameters to access *ResourceMonitor* for real-time personnel locations, as shown in Figure 7.1 (b) (Compare with Figure 6.2 (b)).

Can these threats be thwarted by existing security mechanisms? In the former scenario,

a network-based IDS (e.g., Snort) may be used to examine the HTTP traffic and look for specific patterns (such as SQL quote marks). However, the signatures used are either so generic that they generate an excessive amount of false positives, or so specific that they only work for certain types of applications. Furthermore, it is a labor intensive effort to update the signature database to keep up with evolving attacks. The limitations and low degree of effectiveness of IDS tools against such application-level attacks have been highlighted in recent empirical studies [51]. In the second scenario, since the user is a legitimate user with full access to the system, he will not trigger any network or host based IDS alarms. In this case, the conventional mechanisms are completely ineffective against insider attacks at the application level.

Given these limitations, I believe a more reliable and resilient approach should (a) complement traditional security approaches with additional focus on the application level behavior and (b) use the system's "normal" usage model as the basis for threat detection, which eliminates the need for maintaining attack specifications. The new approach will have the obvious advantage of being effective against insider attacks as well as outside, and being able to detect threats both known or unknown.

7.2 Formalizing the Threat Detection Problem

I first use the definitions and assumptions from Section 6.2 to formally state the detection problem. Given the assumptions of the observability of the UIEs, we can use them to divide the system's execution event traces into itemsets or "baskets", with each basket being a UIE U_i's perceived execution closure or PEC: $U_i^+ = \{< e_1, e_2, ..., e_n >\}$, as illustrated in Figure 7.2. In a multi-user system such as EDS, multiple concurrent user sessions will cause UIEs to overlap. The events a and c (highlighted in yellow), for example, are captured in both U_3^+ and U_4^+. Conceivably, the larger the number of concurrent users, the more overlap UIEs will have with one another. Concurrency of user activities turns out to be the primary source of noise in the event data and thus a major challenge my research needs to overcome.

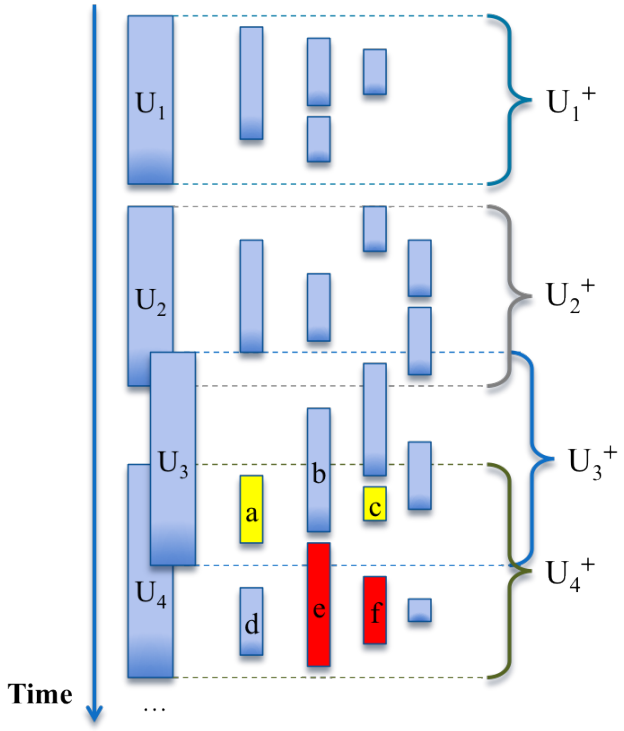


Figure 7.2: Creating PECs from System Execution Traces

When the target system is under attack, anomalous and potentially malicious events may be injected by the attacker and captured in the event stream, like events e and f (highlighted in red) in Figure 7.2. Our detection problem can thus be summarized as the following question:

> *Given an event stream $Q = \{..., e_{j-1}, e_j\}$ in which e_j is the most recent event, the observed UIEs $\{..., U_{i-1}, U_i\}$ and their respective PECs $\{U_i^+\}$, what is the likelihood of e_j being an anomaly?*

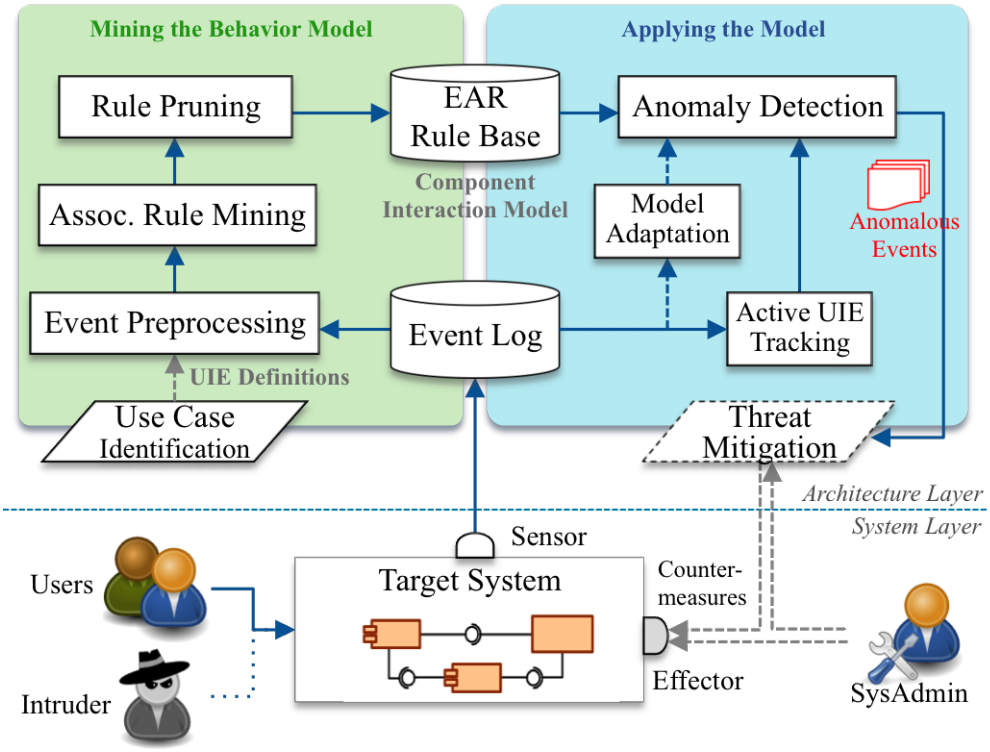


Figure 7.3: ARMOUR Framework Overview

7.3 ARMOUR Framework Overview

The architecture of my mining framework, codenamed ARMOUR, is depicted in Figure 7.3. First we see that the framework follows the FORMS reference architecture (recall Figure 4.1 in Chapter 4), with the architecture layer serving as the meta-level subsystem that protects the base-level system layer. The architecture layer consists of two major phases that form a continuous MAPE-K cycle that monitor and adapt to the target system at runtime:

In the first phase, **mining the behavior model**, system execution traces are captured by sensors on the target system, from which inter-component events are extracted and stored in the event log. The events are then preprocessed to produce itemsets based on user-initiated UIE transactions. The event preprocessing step is informed by the pre-identified UIE types from the Use Case Identification step, which is based on one-time human input (recall my assumption that UIEs are identifiable). Once events are preprocessed

into itemsets, the Apriori algorithm introduced in section 6.3 is applied to the itemsets to generate a collection of event co-occurrence rules. Intuitively, if an event does *not* have any strong associations with past events in the model, the event is likely to be anomalous. As will be discussed later, since the system behavior may vary greatly depending on usage context, I tailor the mining algorithm to be UIE dependent. Since mining may generate a large number of rules, some of which may be invalid and redundant, I prune the generated rules to arrive at a small number of useful rules that can be applied efficiently at runtime. Section 7.4 describes the details of the mining process.

The second phase is **applying the model** on the current system event stream to detect anomalous behavior. I developed an efficient detection algorithm for this purpose, which produces a quantitative anomaly measure for each event. When the anomaly likelihood exceeds a certain threshold, the event is marked as suspicious and its details are recorded. As mentioned earlier, the number of simultaneously running user sessions may fluctuate at runtime and adversely impact the detection accuracy. I therefore take an extra step to monitor the degree of execution concurrency and adapt the detection model accordingly. Once detected, the anomalous events trigger alerts to the system's security administrator or an autonomous agent that would examine them as part of the ongoing threat mitigation process and deploy countermeasures to the target system such as isolating the compromised components. This last step is beyond the scope of anomaly detection but will be the focus of Chapter 9. Section 7.5 elaborate on the detailed design of this phase.

It's worth highlighting that the ARMOUR framework flags an anomalous event along with the potentially comprimised components and encompassing use case(s). These serve as important clues for a system administrator to discover and verify the strategy and *intent* of a malicious attack. In contrast, conventional intrusion detection mechanisms typically can only pinpoint suspicious running processes or network packets. It is often very hard for the system adminstrator to piece together the bigger picture based on these lower level anomalies.

7.4 Mining the Behavior Model using ARM

7.4.1 Event Preprocessing

The first step of my approach is pre-processing the event log into itemsets as input to the mining algorithm. As mentioned in Section 7.2, this essentially involves building PECs for UIEs. For the EDS system, because all user actions are initiated at the user interface component $HQUI$, identifying UIEs is a trivial task. In my study I used a subset of EDS functionality, represented by UIEs such as $HQUI \rightharpoonup SimulationAgent$, $HQUI \rightharpoonup StrategyAnalyzer$, etc.

By reading the start and end timestamps of the UIEs, we can easily keep track of the "active baskets" in the system, and incoming events will fall into one or more of them. When a basket closes, an itemset is produced and stored.

7.4.2 Tailoring the Apriori Algorithm

Unlike traditional associations mining introduced in section 6.3, I use the Apriori algorithm in two distinctive ways. First, given the fact that anomalies, especially malicious attacks seeking to covertly exploit the target system, are rare events occurring with very low frequency, the vast majority of the EARs represent normal system use. Therefore under the threat detection context I set the minimum support level $minsupp$ to very low (e.g., 0.1) in order to comprehensively capture the system behavior model. This is very different from the typical use of ARM, which is to discover only the highly frequent patterns. I choose not to set the minimum confidence level $minconf$, because I want to use the confidence values in the detection algorithm (next section) rather than as a rule elimination mechanism.

Second, we recognize that the prior probabilities of UIEs that drive system behavior may vary greatly. Some user actions, such as viewing map and weather data for situation awareness, occur quite frequently, whereas others such as performing system maintenance occur far less often. Furthermore, the system behavior shifts over time. Deploying disaster response resources, for instance, tends to occur at times of emergency. As a result, it is

likely that some component interaction events, esp. those associated with infrequent user actions, may not meet the minimum support level. This insight made us realize that we need to mine the component interactions *under a specific usage context.* I therefore re-define an EAR to be of a tailored form:

$$X \Rightarrow Y : < supp_i, conf_i \mid UIE_i > \tag{7.1}$$

where $supp_i$ and $conf_i$ are the *conditional* support and confidence levels with respect to a certain UIE type. The implementation of this tailored approach is relatively simple: the training itemsets from the event log are put into different "bins" by their UIE types; the Apriori algorithm is applied to each bin to generate the conditional EARs for the specific UIE (e.g. $HQUI \rightharpoonup SimulationAgent$). The rule base can be viewed as having multiple partitions, one for each UIE type.

7.4.3 Rule Base Pruning

A direct consequence of using very low support setting is that an excessively large number of association rules are produced. Therefore, pruning the rule base and reducing its size becomes a critical component of my approach. Fortunately, a number of heuristics can come to assistance in this regard. The following two, in particular, have shown to be very effective:

1. *Single event pruning heuristic.* Under our problem setting, we would like to detect anomalous events as they occur by filtering the ongoing event stream, one event at a time. As such, the detection algorithm we use only applies to those rules with a single event on the right-hand side (RHS). It can be easily shown that pruning rules with more than one event on the RHS does not result in loss of decision-making information in our problem setting: Suppose we have rule $r_1 : \{a, b\} \Rightarrow \{c, d\}$ in the rule base. We must also have $r_2 : \{a, b, c\} \Rightarrow \{d\}$ and $r_3 : \{a, b, d\} \Rightarrow \{c\}$ in the rule base, because

- r_2 and r_3 have the same support value $\sigma(\{a,b,c,d\})/N$ as r_1, according to equation (6.2);
- r_1's confidence value is $\sigma(\{a,b,c,d\})/\sigma(\{a,b\})$ according to equation (6.3), whereas r_2's is $\sigma(\{a,b,c,d\})/\sigma(\{a,b,c\})$. Because the frequency of events $\{a,b,c\}$ occurring together is equal to or lower than that of its subset $\{a,b\}$, r_2 has an equal or higher confidence value than r_1. Same can be said for r_3. This is known as the anti-monotone property of the confidence metric [172].

Therefore we can use the latter two rules in anomaly detection without having to use the first one. My experimentations showed that as many as 70% of the rules may be pruned by this heuristic alone.

2. *Reverse-ordered EAR pruning heuristic.* This heuristic further stipulates that we can prune any rule $X \Rightarrow Y$ where $\forall e_i \in X \wedge e_j \in Y : e_j.c_{dst} = e_i.c_{src}$. This rule simply predicts an event should have happened in the past, which is not useful for my purposes. For example, the rule

$$\{WeatherAnalyzer \rightharpoonup Weather,\ Weather \rightharpoonup Clock\}$$
$$\Rightarrow \{StrategyAnalyzer \rightharpoonup WeatherAnalyzer\}$$

simply predicts that, when e_5 and e_6 are present, event e_4 should have occurred (see Figure 6.2 (a)). Such rules usually have strong support and confidence values, but does not help explain why e_4 occurs in the first place, therefore they offer no additional value. Such rules can be safely pruned, further reducing the rule base size.

More details of the rules pruning results will be given in section 7.6.4.

7.5 Detecting Anomalous Behavior

Since the generated EARs collectively represent the system's *normal* behavior in the form of component interactions, we have reason to deem any component interaction pattern

that falls outside of this model to be an anomaly. My detection method therefore can be viewed as a form of *infrequent pattern mining*. To phrase it using security nomenclature, my method takes an *anomaly-based* approach for threat detection as opposed to a *signature-based* approach: rather than attempting to formulate and recognize all possible attack signatures, we consider any interaction pattern that falls outside of the normal system usage a potential threat. As will be discussed later in the thesis, the former approach has a significant advantage of being effective against future unknown threats.

I have developed an effective algorithm for quantitatively determining the anaomaly likelihood of an event. The pseudo-code skeleton of the method, `measureAnomaly`, is shown in Algorithm 1.

Algorithm 1 measureAnomaly: Determine Anomaly Likelihood

Input: $\mathcal{R} = \{r_i\}$ ▷ Rule base, use case partitioned, indexed
Input: $minsupp$ ▷ Mininum support level used by Apriori
1: **procedure** MEASUREANOMALY(e)
2: $L_{anomaly}(e) \leftarrow 1$
3: **for** $U_i \in findEnclosingUIEs(e)$ **do**
4: $\{PTP_j\} \leftarrow findPTPs(U_i, e)$ ▷ Find all perceived triggering paths for e
5: $conf \leftarrow MAX_j(ruleLookup(PTP_j, U_i))$ ▷ Find highest confidence
6: $L_{anomaly}(e) \leftarrow L_{anomaly}(e) \times (1 - conf)$
7: **end for**
8: $L_{anomaly}(e) \leftarrow L_{anomaly}(e)^{1/m}$
9: **return** $L_{anomaly}(e)$
10: **end procedure**
11: **procedure** RULELOOKUP($PTP = \{e_1, e_2, ..., e_k\}, UIE$)
12: **if** $\exists r \equiv (\ \{e_1, ..., e_{k-1}\} \Rightarrow \{e_k\}\ : < supp_r, conf_r \mid UIE >) \in \mathcal{R}$ **then**
13: **return** $conf_r$ ▷ Look up rule base, return confidence value
14: **else**
15: **return** $minsupp/2$ ▷ No matching rule found, return estimate
16: **end if**
17: **end procedure**

The idea behind the algorithm is as follows. For any event e captured in the system event log, it may belong to multiple PECs due to system concurrency (recall Figure 7.2), and it is possible to observe the UIE "baskets" of which e is a perceived member, that is, $e \in U_i^+$, $i = 1, ..., m$. Method $findEnclosingUIEs(e)$ (line 3) performs a time

stamp-based search in the event log to find all enclosing UIEs for event e. Within each U_i^+, it is also possible to use event source and destination information to discover zero or more **Perceived Triggering Paths** (PTP) for e, in the form of $S = \{e_1, e_2, ..., e_k, e\}$, such that $e_1.c_{dst} = e_2.c_{src}, e_2.c_{dst} = e_3.c_{src},\ ...\ , e_k.c_{dst} = e.c_{src}$, Method $findPTPs(U_i, e)$ on line 4 achieves this by performing a recursive, depth-first search within the U_i^+ itemset.

As an example, in Figure 7.2, possible PTP's for event c include $\{U_3, c\}$, $\{U_3, a, c\}$, $\{U_3, b, c\}$, $\{U_4, c\}$, and $\{U_4, a, c\}$. Again, the word "perceived" signifies that the sequence is identified purely by observation, not by any knowledge of the events' true causality. Furthermore, because we do not use any temporal information in the association mining context, a PTP may or may not represent a realistic temporal sequence. Intuitively, if we can find a rule of the form $\{U_i, e_1, ..., e_k\} \Rightarrow \{e\}$ in the rule base, it means the presence of event e is "explained" by UIE U_i with a certain confidence level $conf_i$. Conversely, if none of the U_i's can explain e, we have reason to believe it is an anomaly. In other words, the likelihood of e being an anomaly is the conjunctive probability of e ***not*** explained by U_i for all $i = 1, ..., m$. That is, assuming mutual independence of system use case occurrences, the likelihood is $\prod_{i=1}^{m}(1 - conf_i)$ where $conf_i$ is the maximum confidence we can find in the rule base for all PTP's of e within U_i^+ (see line 5). When no matching rules can be found in the rule base, it is necessary that the confidence of a PTP falls between 0 and $minsupp$ (it follows from equations 6.2 and 6.3 that we always have $conf \geq supp$, therefore $minsupp$ serves as a lower bound for confidence values). Therefore the procedure $ruleLookup()$ returns the mean, $(0 + minsupp)/2$, as an estimate for $conf_i$ (line 15).

This likelihood value, however, is not yet a useful detection metric due to the compounding effect introduced by system concurrency. For example, suppose we let $minsupp = 0.10$ and use a detection threshold of 0.9. In a single user scenario, if an anomalous event is injected into the system, its perceived triggering paths will not match any rules in the rule base, therefore procedure $ruleLookup()$ will return the default minimum confidence $minsupp/2 = 0.05$ (line 15 of Algorithm 1). The resulting likelihood of anomaly will then

be $(1 - 0.05) = 0.95 > 0.9$, correctly marking the event as an anomaly. Now let's imagine there are multiple concurrent users in the system and the malicious event happens to fall within 3 concurrent UIE closures. Since the anomaly event will not be "explained" by any of the UIEs, the likelihood will become $\prod_{i=1}^{3}(1 - conf_i) = (1 - 0.05)^3 = 0.86 < 0.9$, falling below the threshold and thus rendering this event as a false negative. To address this challenge, I further normalize the likelihood with regard to the system concurrency at the time of the event by taking its geometric mean:

$$L_{anomaly}(e) = [\prod_{i=1}^{m}(1 - conf_i)]^{\frac{1}{m}} \tag{7.2}$$

where m is the number of enclosing UIEs for e (line 8 of Algorithm 1). Obviously $L_{anomaly}(e) \in [0, 1]$, and the higher its value, the higher likelihood that e is an anomaly. As will be seen in the next section, this metric proves to be quite effective even at high concurrency levels.

Some implementation details of Algorithm 1 are omitted for brevity: Procedure $findEnclosingUIEs(e)$ performs a time stamp-based search in the event log to find all enclosing UIEs for event e; procedure $findPTPs(U_i, e)$ performs a recursive, depth-first search within the U_i^+ itemset to find all PTPs for e. All the EARs in the rule base are indexed using an efficiently encoded "signature" string for each rule, therefore procedure $ruleLookup()$ performs a simple hashtable lookup. Since their logic is straightforward, I have omitted the pseudo code of these procedures.

As mentioned earlier, ARMOUR runs in a continuous loop in parallel to the base-level system. Events are fed into the detection algorithm as they get captured in the event log. At the same time, the rule base is regenerated periodically based on most recent event history. Putting everything together, Algorithm 2 lists the main routine of the ARMOUR framework, omitting details of self-explanatory subroutines.

Algorithm 2 bodyArmour: Main routine

Input: $\mathcal{R} = \{r_i\}$ ▷ Rule base, use case partitioned, indexed
Input: $\mathcal{E} = \{..., e_i, ...\}$ ▷ Event log / queue
Input: $TrainingSize$ ▷ For rule base regeneration
Input: $L_{threshold}$ ▷ Detection threshold

```
1:  procedure BODYARMOUR(E)
2:      e ← dequeue(E)                          ▷ Retrieve event from queue
3:      while e ≠ ∅ do
4:          if refreshPeriodReached() then      ▷ Refresh ruel base
5:              R ← runApriori()
6:              R ← pruneRulebase(R)
7:          end if
8:          L_anomaly(e) ← measureAnomaly(e)
9:          if L_anomaly(e) ≥ L_threshold then
10:             flagEvent(e)                    ▷ Flag event for subsequent mitigation
11:         end if
12:     end while
13: end procedure
```

One final word on setting the detection threshold: similar to choosing confidence levels for rejecting a null hypothesis in statistical tests, ARMOUR users can pick a detection threshold that indicate the level of confidence with which an event can be marked as an anomaly. Just like any other data mining methods, the threshold serves as a trade-off between false positives and false negatives — a higher threshold may reduce false positives but miss some true anomalies, while lowering it produces the opposite effect. There is, however, one caveat that can guide the users in determining a proper threshold: as mentioned earlier, the *minsupp* parameter of the mining algorithm serves as a lower bound for the confidence values $conf_i$ of association rules. With $conf_i \geq minsupp$ and equation 7.2, we have

$$L_{anomaly}(e) \leq [\prod_{i=1}^{m}(1 - minsupp)]^{\frac{1}{m}} = (1 - minsupp) \tag{7.3}$$

therefore $(1 - minsupp)$ serves as an *upper* bound for the detection threshold. Setting the threshold above $(1 - minsupp)$ will result in 0 detections. This also implies that one needs to lower *minsupp* for the mining algorithm in order to increase the upper bound of detection confidence.

7.6 Preliminary Evaluation

7.6.1 Evaluation Setup

The experimentation environment involves an instantiation of the original EDS system that uses Java RMI for sending and receiving event messages among components. As mentioned earlier, logging functionality ensures all component-level events be captured in a MySQL database with synchronized timestamps. A multi-threaded RMI client is written to play back a configurable number of concurrent user sessions. Inter-arrival times of users are exponentially distributed with a mean of 500ms to simulate "busy" usage of the system. Additionally, Gaussian-distributed network latencies are inserted to make the simulations run as close to the real-world environment as possible. Except for the Apriori implementation which was implemented using Weka [71], all other ARMOUR framework components are developed using Java. Both test runs and data analysis are run on quad-core Mac OS X machines.

In order to evaluate the performance of the framework under different concurrency settings, I set up different test cases with a varying number of concurrent users, from 10 to 100. In reality, however, users may have different proficiency levels and browsing habits, therefore the number of concurrent users is not a consistent measure of system concurrency. The number of enclosing UIEs for an event, described in Section 7.5, on the other hand, is a more objective measure. Here I define a concurrency measure γ at the time of event e_k as the moving average of the number of UIE closures to which e_k belongs, computed for the most recent N events:

$$\gamma(e_k) = \frac{1}{N}\left(\sum_{j=k-N+1}^{N} |findEnclosingUIEs(e_j)|\right) \tag{7.4}$$

The correlation between γ with the number of simulated users in my experiments is shown in Figure 7.4 and the header rows of Table 7.1 and subsequent tables. Note that the users in my simulations are active, always-busy users intended to generate a heavy load on the

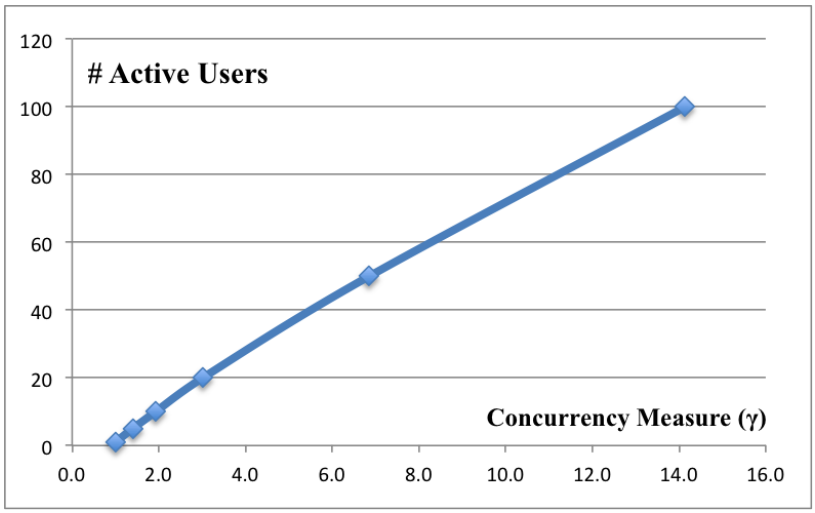


Figure 7.4: Concurrency Measure γ vs. # Users

system, and therefore may represent a much larger number of human users in a real-world setting. The slope of the curve in Figure 7.4 depends on the "activeness" of the users in the system. The near-linear relationship between γ and the number of users can help a system administrator estimate one from the other.

In order to evaluate the ARMOUR performance under realistic usage contexts of the EDS system, I organized the composition of EDS usage into configurable *behavior profiles* that control test runs. As can be seen from three such profiles shown in Figure 7.5, depending on whether the system is "off season", preparing for anticipated emergencies, or during actual emergency response time, the prior probability of UIEs may vary significantly. Under the "emergency time" profile, for example, the "Provide Deployment Advisory" UIE occurs with a 25% probability, while the same event only occurs 5% of the time during off season.

Under each profile, I evaluated ARMOUR using two threat scenarios described in Figure 7.1. Based on the assumption that anomalies are rare events, each of the attack cases was inserted with a probability of $\sim 0.27\%$ (i.e., following the three-sigma rule of thumb for empirical sciences [193]). I will revisit this assumption later in the next chapter. I used a periodic window of 10,000 itemsets from the event log for rule base training, while testing was done on 2,500 itemsets. *minsupp* for the mining algorithm is set to 0.1 in order

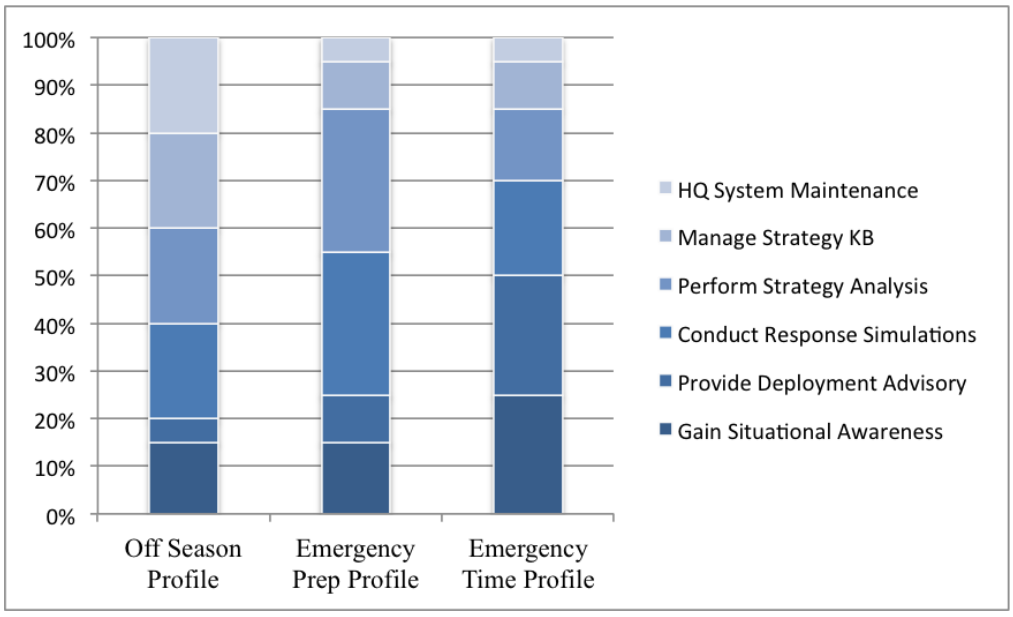


Figure 7.5: EDS User Behavior Profiles

to capture as many normal component interactions as possible. Using $(1 - minsupp)$ as guideline for the upper bound for the detection confidence (recall discussion at the end of Section 7.5), I set the confidence threshold to 0.9.

7.6.2 Threat Detection Accuracy

The test results for the "emergency time" profile are listed in Table 7.1 (A few other profiles were also evaluated with comparable results). We see that, for both scenarios, precision is consistenly above 80% and recall above 70%, proving framework is quite effective against both threats, especially considering these attacks are cover events hidden in a large number of normal user transactions. Furthermore, the results show no accuracy degradation over the 10x increase in system concurrency (in fact they show slight improvements due to the richer model, as indicated by the rule count increase), validating my use of the normalized anomaly measure defined in Section 7.5.

Table 7.1: Detection Results for 2 Threat Scenarios

#active users	10	20	50	100
Concurrency (γ)	5	11	28	56
Threat Scenario A (Figure 7.1 (a))				
TP Count	50	47	40	31
FP Count	12	9	5	0
FN Count	12	18	13	16
TN Count	35,054	34,505	34,398	36,042
TP Rate (TPR)	0.806	0.723	0.755	0.660
FP Rate (FPR)	3.42E-4	2.61E-4	1.45E-4	0.0
Precision	**0.806**	**0.839**	**0.889**	**1.0**
Recall	**0.806**	**0.723**	**0.755**	**0.660**
F-Measure	0.806	0.777	0.816	0.795
EAR Rule Count	78,436	232,992	451,578	1,059,378
Threat Scenario B (Figure 7.1 (b))				
TP Count	57	39	36	46
FP Count	14	7	2	0
FN Count	11	10	15	18
TN Count	33,678	34,398	35,004	34,645
TP Rate (TPR)	0.838	0.796	0.706	0.719
FP Rate (FPR)	4.16E-4	2.03E-4	5.71E-5	0.0
Precision	**0.803**	**0.845**	**0.947**	**1.0**
Recall	**0.838**	**0.796**	**0.706**	**0.719**
F-Measure	0.820	0.821	0.809	0.836
EAR Rule Count	79,658	233,846	305,166	710,936
$TPR = TP/(TP + FN); FPR = FP/(FP + TN)$ $Precision = TP/(TP + FP); Recall = TPR$ $F\text{-}Measure = 2TP/(2TP + FP + FN)$				

7.6.3 Effectiveness of Context-Sensitive Mining

Recall equation 7.1 in Section 7.4 that we tailored the definition of EARs to be conditional on the UIEs, as a way to isolate the component interaction model from usage context variations both across different users and over time. The results in Table 7.1 validated the effectiveness of this *context-sensitive* association mining approach. For comparison, Table 7.2 shows the detection results for threat scenario A under identical configuration settings except for using the plain, unaltered Apriori algorithm. We see that even though recall remains at the same level, precision drops precipitously to about 3% due to a much higher

Table 7.2: Detection Results Using Plain Apriori

#active users	10	20	50	100
Concurrency (γ)	5	11	28	56
Threat Scenario A				
TP Count	52	43	43	31
FP Count	**1,894**	**1,438**	**1,224**	**1,508**
FN Count	14	16	17	16
TN Count	32,632	33,689	33,025	35,288
TP Rate (TPR)	0.788	0.729	0.723	0.660
FP Rate (FPR)	0.055	0.041	0.036	0.021
Precision	0.027	0.029	0.034	0.039
Recall	0.788	0.729	0.723	0.660
F-Measure	0.052	0.056	0.065	0.075

Table 7.3: Rule Pruning Effectiveness

#active users	10	20	50	100
Concurrency (γ)	5	11	28	56
Total Rules Generated	78,782	233,224	408,606	1,523,264
− Heuristics #1 Match	70,567	215,891	380,271	1,452,598
− Heuristics #2 Match	2,862	6,861	10,778	28,929
= Rules Remaining	**5,353**	**10,472**	**17,557**	**41,737**
Pruning Effectiveness	93.2%	95.5%	95.7%	97.3%

number of FPs — many valid events with a lower prior probability are mistakenly tagged as anomalies.

7.6.4 Rule Pruning Effectiveness

As discussed in Section 7.4.3, I have taken steps towards aggressively pruning unnecessary rules from the output of the Apriori algorithm. Table 7.3 shows the effectiveness of the two major pruning heuristics introduced earlier. We see that over 90% of the rules can be safely eliminated. Note that, because the rule base is stored in a hash table and the lookup time is $O(1)$, rule pruning does not boost detection speed but rather serves as a space-saving technique.

Table 7.4: Computational Efficiency Metrics (ARM Method)

#active users	10	20	50	100
Concurrency (γ)	5	11	28	56
Apriori Mining Time (ms)	23,028	44,776	116,784	196,520
Detection Time per Event (ms)	357	476	857	3,362

7.6.5 Computational Efficiency

The running time for the Apriori algorithm largely depends on the number of rules produced. For the detection phase, a quick complexity analysis of Algorithm 1 shows that:

- Running time of $findEnclosingUIEs(e)$ on line 3 depends on concurrency measure γ, i.e., in $O(\gamma)$ time;
- Similarly, the for loop on lines 3-7 is repeated γ times;
- $findPTPs()$ on line 4 carries out a DFS search in U_i^+. Its runtime depends on the size of U_i^+ which is UIE specific, but can be amortized to $O(\gamma)$ time;
- procedure $ruleLookup()$ is hash table-based and runs in $O(1)$.

Therefore we can conclude that Algorithm 1 runs in $O(\gamma)$ time, that is, proportional to the concurrency measure.

Table 7.4 lists the average mining time and detection time per event at different concurrency levels. We can see that the mining time is quite long (may take minutes) but acceptable since it runs periodically in a separate thread. Detection time, however, is problematic. At high concurrency levels, it may take up to a few seconds to process a single component interaction event, which is far too slow and will not keep up with the target system.

7.7 ARM Challenges

In this chapter I have proposed a context-sensitive mining framework that involves monitoring component-based interaction events, generating and pruning a set of probabilistic association rules from system execution history as a component interaction model, and applying the model with an adaptive detection algorithm to efficiently identify potential malicious events.

My evaluation of the approach against a real Emergency Deployment System has demonstrated encouraging results with high detection accuracy, regardless of system concurrency levels, but a few problems still remain.

First and foremost, due to the nature of the association mining technique, our generated architecture model only captured the co-occurrence of components in the user transactions and ignored the temporal information that is inherent in the system execution history. The temporal ordering of component interaction events, even if not entirely accurate due to network time sync errors, can provide valuable insight to the system behavior model at runtime.

As mentioned in Section 7.4, our unique use of the ARM technique to capture the normal system behavior requires us to keep the minimum support level *minsupp* very low. This, coupled with the fact that we have chosen to ignore the temporal information associated with the event sequences, results int the rapid exponential growth in the rule base size. In fact, Table 7.1 shows that for 100 concurrent users, the rule base has grown to over 1 million rules, many of which are extraneous and unnecessary. Even with effective pruning techniques and space-saving tactics, it would be very difficult and impractical for the framework to support large-scale real-world systems with many components, as evidenced by the computational efficiency metrics.

In the next chapter I will enhance the ARMOUR framework with a better mining technique, GSP mining, that addresses this challenge.

Chapter 8: ARMOUR Framework Enhanced: the GSP Approach

In this chapter I describe how sequetial pattern mining, introduced in Section 6.4, can address the challenges with associations mining and enhance the efficiency of the ARMOUR framework, making it more robust for practical use.

8.1 ARMOUR Framework Revisited

The revised architecture of ARMOUR is depicted in Figure 8.1. We see that the overall layout of the architecture layer remains largely the same, consisting of two major phases, *Mining the Behavior Model* and *Applying the Model*.

The major changes happen in the mining phase, with the new building blocks shown in blue in the diagram. At the Event Preprocessing stage, events from the event log are extracted to generate event sequences (as opposed to itemsets) for each UIE based on pre-identified UIE types. Afterwards, a GSP mining algorithm is applied to the sequences. In the revised framework I employ it to build a *pattern registry* that represents a component interaction model for the normal system behavior at runtime. Since regular GSP mining may still generate a large number of patterns, I customize it using the architecture knowledge in order to drastically prune the search space and make the algorithm efficient enough for runtime use, as described in detail in Section 8.2.

The second phase, *applying the model*, remains structurally the same; it is still centered around a detection algorithm that produces a quantitative anaomaly measure for each event. However I do need to make adjustments on the algorithm to work with the GSP patterns. Here I also added an important logic in the main routine so that ARMOUR can determine the optimal training data size. Section 8.3 elaborates on the detailed modifications.

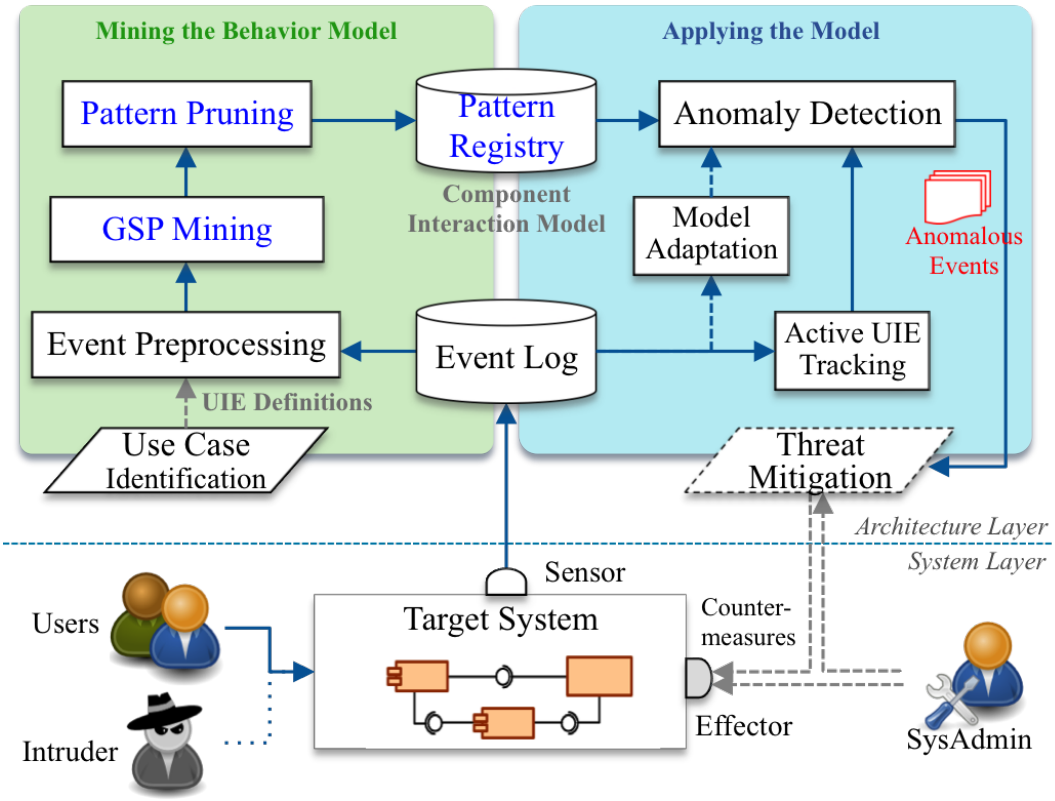


Figure 8.1: Revised ARMOUR Framework based on GSP Mining

8.2 Tailoring the GSP Algorithm

8.2.1 Event Preprocessing

Recall from Section 6.4 that the objective of sequencial pattern mining is to find all subsequences that meet a user-defined minimum support (*minsupp*) among a large number of data sequnces.

The first step of my approach is pre-processing the event log into data sequences as input to the mining algorithm. Under the GSP context, the list of events supporting a user transaction naturally constitutes a data sequence, with each sequence being initiated by a UIE, bounded by its PEC, and ordered by their start time t_s (recall definitions from Section 6.2). For the EDS system, because all user actions are initiated at the user interface component $HQUI$, each UIE is of the form $HQUI \rightharpoonup c \in C$, such as $HQUI \rightharpoonup SimulationAgent$,

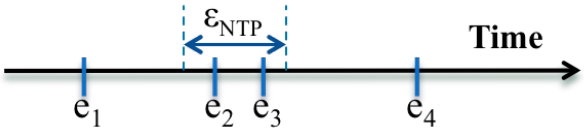


Figure 8.2: Event Sequence Example

$HQUI \rightharpoonup StrategyAnalyzer$, etc. At runtime, we can easily keep track of active user sessions by monitoring web server logs, and allocate each incoming event into one or more UIE clousures according to their start and end timestamps. When a UIE is completed, a data sequence is produced and stored.

Now that we are incorporating temporal information in our model, we need to have a more careful treatment on timestamps. As seen in Section 6.2, concurrent user activities may produce component interaction events that occur at the same time. Further, due to the margin of error in network time, it is difficult to determine the exact order of events occurring with close proximity to one another. Therefore, in the preprocessing step I introduce a sliding ε_{NTP} window: consecutive events whose start timestamps fall within the window are treated as co-occurring and added to the sequence as a single itemset. For instance, the events shown in Figure 8.2 will be sequenced into $\prec e_1, (e_2\ e_3), e_4 \succ$. The ε_{NTP} is configurable to reflect the actual networking environment of the system (LAN, WAN, etc.).

8.2.2 Customized GSP Mining

Recall from Section 6.4 that I use GSP to discover *Event Association Patterns* (EAP) of the form $P = \prec s_1, s_2, ..., s_n \succ :\ supp$ where each element s_i is an itemset of co-occurring events and *supp* is the support value of the pattern.

I need to tailor the GSP algorithm in two unique ways, much like how I customized the Apriori algorithm in the previous chapter. First, as explained in Section 7.4.2, I need to keep the minimum support level *minsupp* to very low (e.g., 0.1) since the vast majority of the EAPs represent normal system use. This is very different from the typical use of

GSP, which is to discover only the highly frequent sequences. A direct consequence of this is that an exponentially large number of pattern sequences will be produced for any slightly more complex system, with unacceptable time and space requirements. Fortunately, our architecture-based heuristic knowledge of the system comes to assistance once again in this regard. We notice that component interactions have causal relationships: an event is triggered by a use case (UIE), and the event's destination component in turn triggers events to other components. Therefore I inserted the following heuristic to the candidate generation phase of GSP: keep a candidate sequence $\prec s_1, s_2, ..., s_n \succ$ *iff* each non-UIE event in the sequence has a preceding triggering event. Formally, for any element s_i,

$$\forall e_v \in s_i \ (\ \exists e_u \in s_{j,1 \leq j \leq i-1} \ (\ e_u.c_{dst} = e_v.c_{src} \ )) \tag{8.1}$$

With this heuristic, the number of added candidates in each GSP iteration becomes linear to the number of possible event types (i.e., source to target component combinations), or $O(|C|^2)$. Since GSP usually runs in a limited number of iterations, this heuristic effectively reducing the growth of candidate sequences from permutational to polinomial in relation to the number of system components.

Second, just like how I accounted for the varying prior probabilities of use cases and potential user behavior shifts over time in Section 7.4.2, I tailor the GSP logic to mine the component interactions *under a specific usage context*. I likewise revise the definition of an EAP:

$$P = \prec s_1, s_2, ..., s_n \succ : \ supp_i \mid UIE_i \tag{8.2}$$

where $supp_i$ is the *conditional* support with respect to a specific use case. The pattern registry can thus be viewed as having multiple partitions, one for each UIE type.

8.3 Detecting Anomalous Behavior

Our anomaly detection strategy hasn't changed from the ARM-based approach in the previous chapter: Since the generated EAPs collectively represent the system's *normal* behavior, we have reasons to suspect any component interaction pattern that falls outside of this model to be an anomaly.

The revised anomaly detection algorithm based on the GSP technique instead of ARM is shown in Algorithm 3.

Algorithm 3 measureAnomaly: Determine Anomaly Likelihood (Revised)

Input: $\mathcal{P} = \{p_i\}$ ▷ Pattern registry, use case-partitioned, indexed
Input: $minsupp$ ▷ Minimum support level used by GSP
1: **procedure** MEASUREANOMALY(e)
2: $L_{anomaly}(e) \leftarrow 1$
3: **for** $U_i \in findEnclosingUIEs(e)$ **do**
4: $\{PTS_j\} \leftarrow findPTSs(U_i, e)$ ▷ Find all perceived triggering sequences for e
5: $supp_i \leftarrow Max_j(patternLookup(PTS_j, U_i))$ ▷ Find highest support
6: $L_{anomaly}(e) \leftarrow L_{anomaly}(e) \times (1 - supp_i)$
7: **end for**
8: $L_{anomaly}(e) \leftarrow L_{anomaly}(e)^{1/m}$
9: **return** $L_{anomaly}(e)$
10: **end procedure**
11: **procedure** PATTERNLOOKUP(PTS, UIE)
12: **if** $\exists\ p \equiv (PTS : supp_p \mid UIE) \in \mathcal{P}$ **then**
13: **return** $supp_p$ ▷ Look up pattern registry, return support value
14: **else**
15: **return** $minsupp/2$ ▷ PTS not found in registry, return estimate
16: **end if**
17: **end procedure**

Even though the idea behind the algorithm is the same, I needed to make the following modifications so *measureAnomaly*() can work with the EAPs instead of the EARs:

- First, I replaced the method $findPTPs()$ in Algorithm 1 with the new method $findPTSs()$. Since the orignal algorithm is based on the association rule base without any temporal information, I constructed the logical execution sequences

called Perceived Triggering Paths (PTP) within each PEC U_i^+. With the registry of EAPs, we can now use the source and destination components of the events *and* their temporal order to discover zero or more *Perceived Triggering Sequences* (PTS) for e, in the form of $\{PTS_e = \prec U_i, e_1, ..., e_k, e \succ\}$, such that $U_i.c_{dst} = e_1.c_{src}, e_1.c_{dst} = e_2.c_{src}, \ ... \ , e_k.c_{dst} = e.c_{src}$. Method $findPTSs(U_i, e)$ on line 4 achieves this by performing a recursive, depth-first search within U_i^+. Again, the word "perceived" signifies that the sequence is purely by observation, not by any knowledge of the events' true causality. For example, in Figure 7.2, potential PTS's for event c include $\prec U_3, a, c \succ$, $\prec U_3, b, c \succ$, $\prec U_3, c \succ$, $\prec U_4, a, c \succ$, and $\prec U_4, c \succ$.

- Second, in Algorithm 1 I computed the anomaly likelihood with the confidence values $conf_i$ found in the rule base. In the new algorithm we only need to work with the support values of the EAPs found in the registry, $supp_i$. The likelihood of e being an anomaly is still the conjunctive probability of e ***not*** explained by U_i for all $i = 1, ..., m$. Assuming mutual independence of system use case occurrences, the anomaly likelihood is $\prod_{i=1}^{m}(1 - supp_i)$, where $supp_i$ is the highest support we can find in the pattern registry for e's triggering sequences within U_i^+ (see lines 5 and 6). Note that we still normalize the likelihood value by taking its geometric mean:

$$L_{anomaly}(e) = [\prod_{i=1}^{m}(1 - supp_i)]^{\frac{1}{m}} \tag{8.3}$$

  where m is the number of enclosing UIEs for e (line 8 of Algorithm 3). Obviously $L_{anomaly}(e) \in [0, 1]$, and the higher its value, the higher likelihood that e is an anomaly.

- Finally, I replaced the *ruleLookup*() method with *patternLookup*(), which looks up the pattern registry for the given pattern with respect to the specific UIE type and returns its support value. When no matching EAP is found, it is necessary that the support of a PTS falls between 0 and *minsupp*. In this case *patternLookup*() returns

the mean, $(0 + minsupp)/2$, as an estimate for $supp_i$ (line 15).

As mentioned in Section 7.5, ARMOUR runs in a continuous loop in parallel to the target system. Just like the rule base, the pattern registry also needs to be regenerated periodically based on most recent event history to reflect the latest system behavior. But one critical question still remains that hasn't been addressed in the previous chapter: how can we tell the EAP registry (or the rule base, for that matter) is a good enough capture of the current system's normal behavior? Through experimenting with the GSP tests over EDS system event traces, I observed that as the size of the training set increases, the size of the pattern registry $|P|$ grows asymptotically towards a stable limit, indicating nearly all component interactions have been captured. Accordingly, I further revise the main routine of the ARMOUR framework to run GSP repeatedly over increasing number of training sequences from the recent execution history, until the increase in $|P|$ over the previous iteration falls below a threshold (say, 5%). The importance of this step is two-fold: it helps determine the optimal training window size, but also serves as a tell-tale sign that the system behavior model is stable and therefore the framework is effective. Conversely, if $|P|$ fails to converge over an extended training period, it indicates the system behavior may be in a fluctuating state and the effectiveness of the model may be limited.

Algorithm 4 lists the revised main routine of the ARMOUR framework, omitting details of self-explanatory subroutines. Lines 4-12 reflect the newly added logic for determining optimal training size, which runs in separate threads that do not block the continous detection process. Please note the complementary relationship between the detection threshold and the *minsupp* parameter (line 7) hasn't changed: the higher confidence we would like on the detections, the lower support level we need for the GSP mining algorithm in order to have a more thorough capture of the system behavior model.

Algorithm 4 bodyArmour: Main routine (Revised)

Input: $\mathcal{P} = \{p_i\}$ ▷ Pattern registry, use case-partitioned, indexed
Input: $\mathcal{E} = \{..., e_i, ...\}$ ▷ Event log / queue
Input: $L_{threshold}$ ▷ Detection confidence threshold

procedure BODYARMOUR($\mathcal{E}$)
  $e \leftarrow dequeue(\mathcal{E})$ ▷ Retrieve event from queue
  while $e \neq null$ **do**
    if $refreshPeriodReached()$ **then** ▷ Refresh pattern registry
      $trainingWindow \leftarrow 100$
      repeat [in new thread] ▷ Determine training window
        $minsupp \leftarrow 1.0 - L_{threshold}$
        $\mathcal{P}_{new} \leftarrow runCustomGSP(minsupp)$ ▷ Run in new thread
        $trainingWindow \leftarrow trainingWindow + 100$
      until $\Delta(|\mathcal{P}_{new}|) < 0.05$
      $\mathcal{P} \leftarrow \mathcal{P}_{new}$
    end if
    $L_{anomaly}(e) \leftarrow measureAnomaly(e)$
    if $L_{anomaly}(e) \geq L_{threshold}$ **then**
      $flagEvent(e)$ ▷ Flag event for subsequent mitigation
    end if
  end while
end procedure

8.4 Evaluation

The evaluation of the revised ARMOUR framework follows the same setup described earlier in Section 7.6.1, so it's not repeated here.

The only additional configuration parameter is for network timekeeping: As mentioned in Section 8.2.1, we take into account the impact of network timing errors as we pre-process the event log into data sequences. Even though we conducted all test runs within a LAN with sub-millisecond NTP error margins, we set ε_{NTP} to be 10ms, a value safe for the public Internet [177] in order to mimic the real-world EDS runtime environment.

8.4.1 Determining Training Set Size

As mentioned in Section 8.3, I incorporated logic in the ARMOUR main routine to run GSP repeatedly over an increasing training window size, until the growth of the EAP registry stablizes.

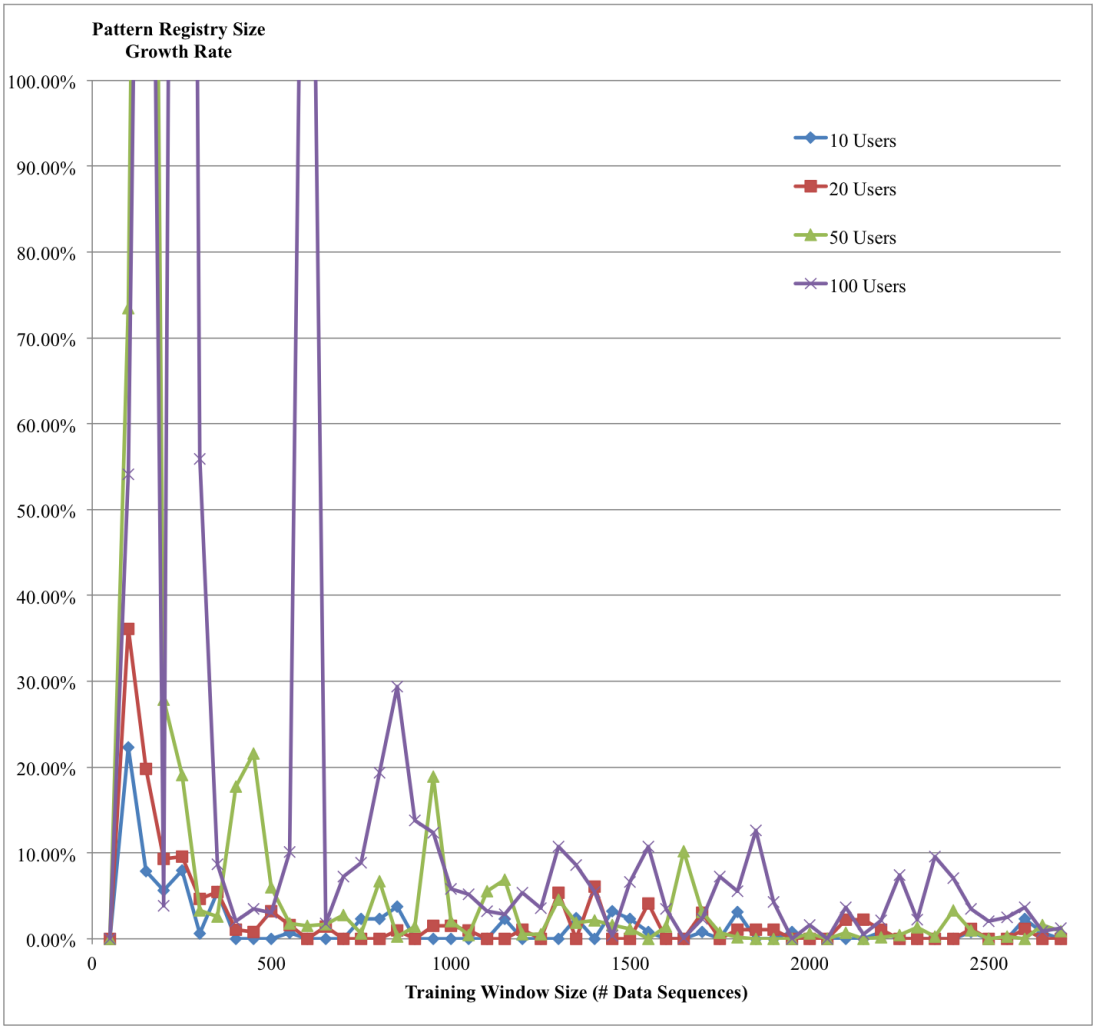


Figure 8.3: Pattern Registry Growth Over Training Window Size

Figure 8.3 shows the pattern registry size (i.e., EAP count) growth rate over the previous iteration under different concurrency settings. It confirms our intuition that as more sequences from the system execution history are used in the mining process, the EAP count increases but the growth rate asymptotically reduces toward zero, when most of the EAPs for normal system behavior are discovered. It also shows that the higher the concurrency level, the more noise is added in system behavior (recall Section 7.2), therefore more training data is needed for the pattern registry to converge (with a training window of about 2,500 sequences under 100 concurrent users).

8.4.2 Threat Detection Accuracy

My evaluation was done under the "emergency time" profile using the same two threat scenarios described in Figure 7.1, with the attack cases inserted with a probability of 0.27% (i.e., following the three-sigma rule as before). I will revisit this assumption later in this section. I used a training window of 2,500 sequences (that is, UIEs) from the event log for GSP mining, while testing was done on 1,000 sequences. *minsupp* for the mining algorithm is set to 0.1 in order to capture as many normal component interactions as possible. Accordingly I set the confidence threshold to $(1 - minsupp)$ (recall line 7 of Algorithm 4), or 0.9.

The test results, listed in Table 8.1, demonstrate my revised framework is very effective in detecting both threats hidden among a large number of normal system events, especially considering these attacks may have been missed by IDS sensors. Precision is close or equal to 100% for most tests, higher than the ARM-based method; recall is consistently over 70%, at the same level as the ARM-based method. The results still show no degradation over the 10x increase in system concurrency, an indicator of not only the effectiveness of the concurrency-normalized anomaly measure defined in Section 8.3, but also the practical potential of using the ARMOUR framework with concurrent, multi-user systems.

The most conspicuous difference compared with the ARM-based method, however, is in the size of the model: The EAP counts as listed in the table are in the 10^2 range, orders of magnitude smaller than the number of EARs captured from the associations mining-based model! We now clearly see that the temporal information of events has helped eliminate numerous extraneous rules, resulting in a much leaner and efficient model. The efficiency gains will become evident later in the performance analysis section. Further, because much noise has been eliminated through the temporal sequence mining process, we also no longer see any significant growth in the EAP count as system concurrency increases 10-fold, making the model more scalable and more resilient to system concurrency fluctuations.

Having validated the effectiveness and advantages of GSP Mining, I now proceed to more extensive evaluations of the ARMOUR framework.

Table 8.1: Detection Results for 2 Threat Scenarios: GSP-based

#active users	10	20	50	100
Concurrency Measure(γ)	5	11	28	56
Threat Scenario A (Figure 7.1 (a))				
TP Count (#Events)	14	22	16	26
FP Count	2	0	0	0
FN Count	5	9	6	9
TN Count	14,375	13,606	13,377	13,954
TP Rate (TPR)	0.737	0.710	0.727	0.743
FP Rate (FPR)	1.39E-4	0.0	0.0	0.0
Precision	**0.875**	**1.0**	**1.0**	**1.0**
Recall	**0.737**	**0.710**	**0.727**	**0.743**
F-Measure	0.800	0.830	0.842	0.852
EAP Count	125	147	147	152
Threat Scenario B (Figure 7.1 (b))				
TP Count	17	19	14	15
FP Count	1	0	0	0
FN Count	5	8	4	4
TN Count	13,475	14,005	13,697	14,810
TP Rate (TPR)	0.773	0.704	0.778	0.789
FP Rate (FPR)	7.42E-5	0.0	0.0	0.0
Precision	**0.944**	**1.0**	**1.0**	**1.0**
Recall	**0.773**	**0.704**	**0.778**	**0.789**
F-Measure	0.850	0.826	0.875	0.882
EAP Count	127	143	147	147
$TPR = TP/(TP + FN); FPR = FP/(FP + TN)$ $Precision = TP/(TP + FP); Recall = TPR$ $F\text{-}Measure = 2TP/(2TP + FP + FN)$				

8.4.3 Training Environment (or the Elimination Thereof)

Under ideal circumstances, the pattern registry would need to be mined using clean, attack-free training sets before use at runtime on real data. In fact many prior data mining techniques for security require such supervised learning (more details in Section 11.2). I would then periodically re-train the framework as the system's usage behavior evolves. The nature of the pattern mining process, however, hinted that *we can eliminate the need for a clean training environment altogether* – due to the outlier nature of anomalies, they do not occur frequently enough to make their way into the EAPs, as long as the *minsupp* level of

the GSP algorithm is set well-above the anticipated anomaly rate.

It is worth noting that the detection results shown in Table 8.1 were indeed produced while running ARMOUR in parallel to the EDS system, with the model trained over actual, tainted event logs. This confirms that ARMOUR can run *unsupervised* and does not need a clean training environment[1]. This gives ARMOUR a big practical advantage as clean training data for a real-world system is usually difficult to obtain.

8.4.4 Detecting Unknown Threats

As alluded to in Section 7.5, because an anomaly-based detection model focuses on capturing normal system behavior rather than capturing all possible attack signatures, it has the natural advantage of being able to detect new threats that are not previously known. To validate that the ARMOUR framework has this capability, I set up a cross-validation procedure that performs GSP pattern mining in the presence of Attack A and then runs threat detection against an event log that includes injected Attack B, and vice versa.

Table 8.2 shows the test results at different concurrency levels. We can see the algorithm is equally effective for detecting threats that were not present during the mining phase! Given the ever-changing attack vectors and so-called "zero-day" threats for applications today, ARMOUR can be an extra layer of protection against future exploits involving undiscovered software vulnerabilities.

8.4.5 Sensitivity Analysis

ARMOUR makes very few assumptions on the target system's runtime behavior and its runtime environment. In fact, the only two key parameters my evaluations depend on are the *minsupp* level of the mining algorithm (set to 0.1) and the prior probability of anomalies (set to below 0.27%). I would like to examine how ARMOUR's detection performance is impacted by these parameters in order to discover any limitations of my approach.

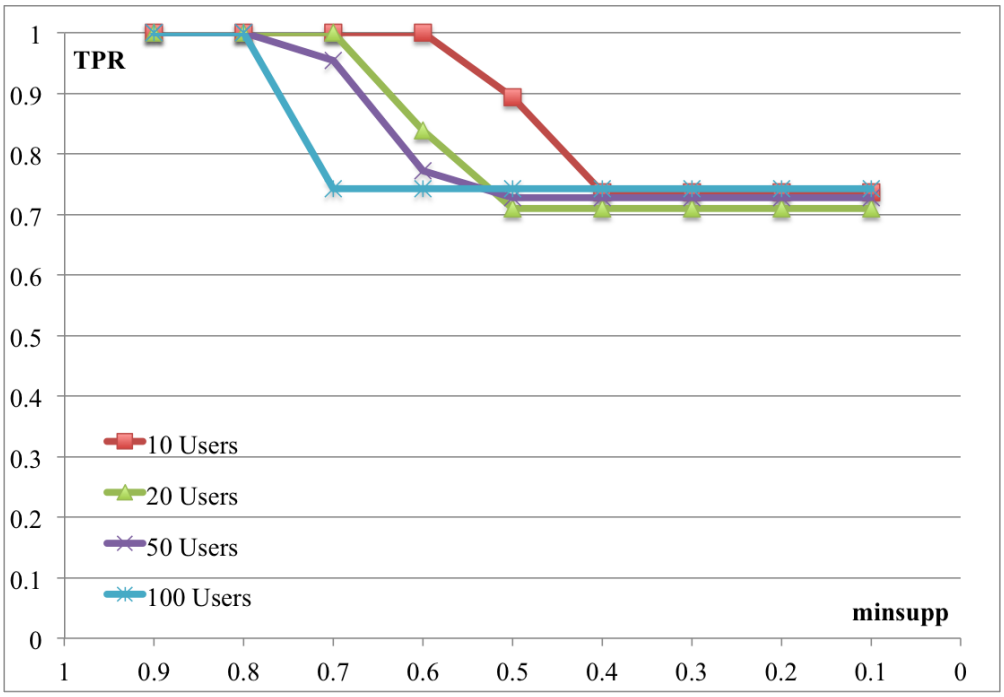
[1]Some may argue my approach is *semi-supervised* since we have a notion of a normal class, however unsupervised is probably more appropriate since we do not have any definitive label for normal events, only a probabilistic estimate

Table 8.2: Detection Results for Unknown Threats

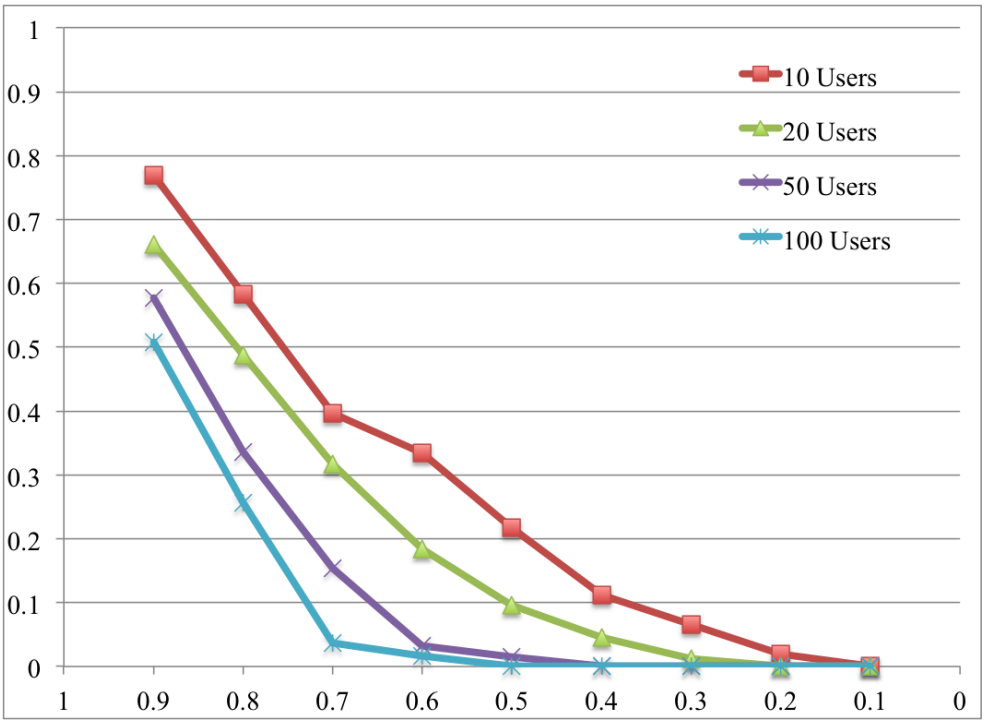
#active users	10	20	50	100
Concurrency(γ)	6	11	28	56
Mining with Attack A, detection against Attack B				
TP Count (#Events)	13	15	19	10
FP Count	1	0	0	0
FN Count	5	5	8	2
TN Count	14,444	14,239	14,090	14,103
TPR	0.722	0.750	0.704	0.833
FPR	6.92E-5	0.0	0.0	0.0
Precision	0.929	1.0	1.0	1.0
Recall	0.722	0.750	0.704	0.833
F-Measure	0.813	0.857	0.826	0.909
Mining with Attack B, detection against Attack A				
TP Count	16	24	14	16
FP Count	1	0	0	0
FN Count	8	12	7	6
TN Count	14,613	14,139	13,996	13,855
TPR	0.667	0.667	0.667	0.727
FPR	6.84E-5	0.0	0.0	0.0
Precision	0.941	1.0	1.0	1.0
Recall	0.667	0.667	0.667	0.727
F-Measure	0.780	0.800	0.800	0.842

First, I repeated the test runs for Attack A under different *minsupp* levels, from high to low, in order to see the impact on the True Positive Rate (TPR) and False Positive Rate (FPR) results, which are shown in Figure 8.4. We see that as *minsupp* is lowered, TPR remains at a high level (above 70%) and is not very sensitive to minsupp changes, while FPR rapidly decreases towards 0. These plots confirm our earlier understanding that setting support at low levels can effectively capture the component interaction model. The "hockey stick" shape of the FPR plot indicates the framework is far less sensitive to *minsupp* at lower levels, implying that there is no need to search for an optimal *minsupp*; ARMOUR can operate effectively over a range of *minsupp* towards the lower end.

Also of interest to my work is the prior probabilities of anomalous events, which we assumed is very low compared with normal system use. My approach, especially when run



(a) TPR Plot



(b) FPR Plot

Figure 8.4: TPR and FPR Sensitivity to *minsupp*

Table 8.3: Detection Results Under Increased Anomaly Rates

Anomaly Rate	0.27%	1.0%	2.0%	3.5%	5.0%
TP Count	22	5	11	11	6
FP Count	0	2	0	1	1
FN Count	**9**	**28**	**59**	**117**	**169**
TN Count	13,606	13,892	15,066	14,712	14,284
TPR	0.710	0.152	0.157	0.086	0.034
FPR	0.0	1.44E-4	0.0	6.80E-5	7.00E-5
Precision	1.0	0.714	1.0	0.917	0.857
Recall	0.710	0.152	0.157	0.086	0.034
F-Measure	0.830	0.250	0.272	0.157	0.066

in the unsupervised mode with tainted data, is based on the premise that rare events do not occur frequent enough to interfere with building the normal system usage model. Therefore, it is important to understand ARMOUR's sensitivity to the prior probability of anomalous events.

Using the Attack A scenario, I performed test runs that gradually increased the anomaly rate above the default 3-sigma level, with other parameters kept the same. The results for 20 concurrent users are listed in Table 8.3. The results of default anomaly rate (0.27%) are repeated from Table 8.1 for comparison. We can see that, while precision remains high, recall deteriorates quickly when the anomaly rate gets higher, driven by more false negatives, indicating more and more anomalies start to be considered normal events.

The results are hardly surprising: when the prior probability of anomalies approaches *minsupp*, anomalous patterns start to be captured by the mining algorithm as EAPs and thus become mixed with the normal system behavior. At this point the model starts to lose its ability to distinguish anomalous events from legitimate ones[2]. Fortunately, higher-frequency attacks that seek to do immediate harm to the system will likely be recognized as Denial of Service (DoS) attacks and dealt with accordingly. This serves as a good reminder that ARMOUR should be used in conjunction with existing intrusion detection mechanisms rather than replacing them.

[2]Clearly, if clean training data is available from a controlled environment, this will not be an issue

8.4.6 Computational Efficiency

Mining Performance

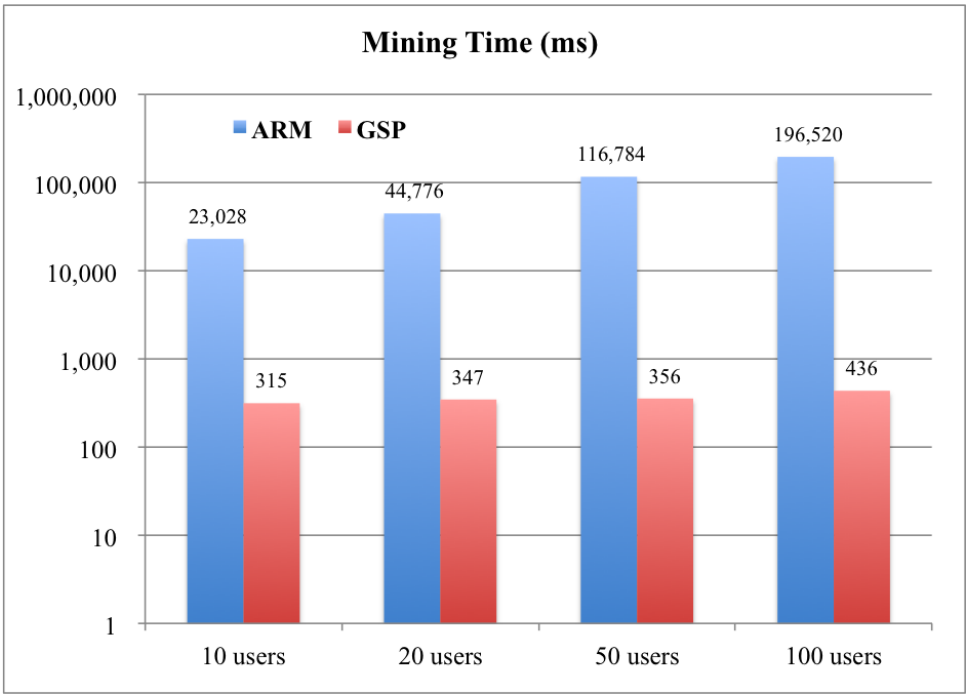
Figure 8.5 (a) shows the time it takes for mining the architecture model for both ARM and GSP under threat scenario A. Note the logarithmic scale. Since the complexity for both algorithms is closely tied to the number of valid candidates they generate, we expected to see a strong correlation between the EAR/EAP count and the mining time, as confirmed by the plot: Because the tailored GSP algorithm drastically reduced the numbrer of valid patterns, the mining time is also orders of magnitude shorter than that of the ARM algorithm.

For GSP in particular, the plot shows that at the highest concurrency level (100 users), it takes 436 milliseconds to complete a mining run. Considering the mining process runs in a separate thread and the pattern registry only needs to be refreshed periodically, it is quite practical to apply our algorithm to support on-the-fly anomaly detection at runtime. To put this in perspective, I was not even able to complete a mining run of the original, unaltered GSP algorithm for just 10 concurrent users once *minsupp* drops below 0.4 due to either time or memory limitations; the number of candidates it generated was simply too large. The GSP algorithm was not designed to be a main-memory algorithm to begin with, and making multiple passes of large numbers of pattern candidates in external storage will simply take too long. By comparison, our architecture-based heuristics prove to be highly effective in reducing the candidate search space even at low support levels.

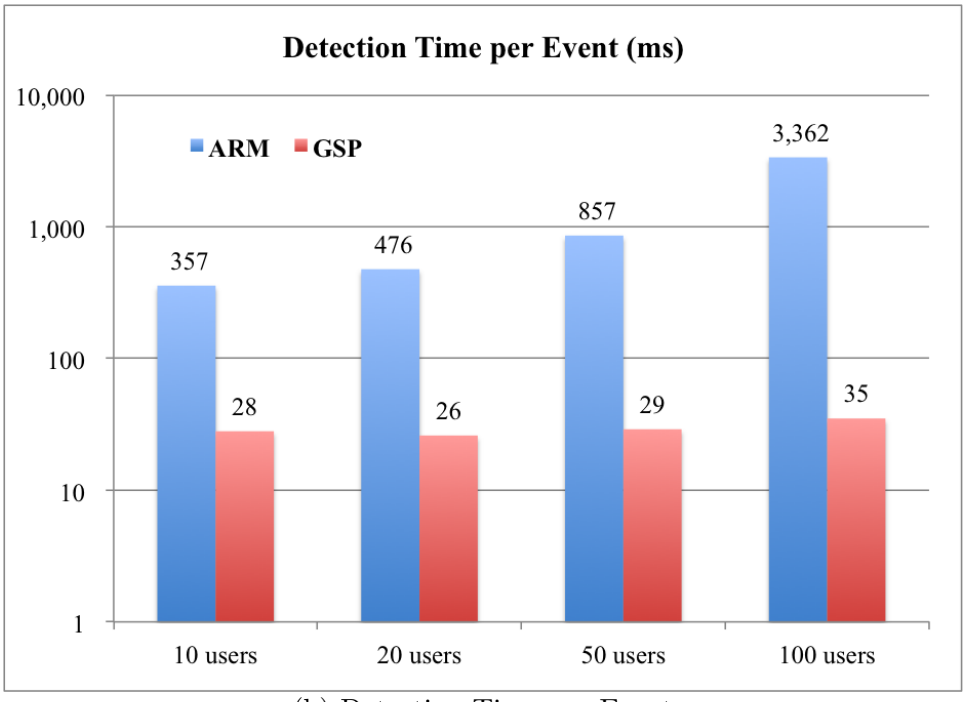
Anomaly Detection Performance

To be used at runtime, the detection portion of ARMOUR must be efficient and fast, in order to keep up with real-time system execution. A quick complexity analysis of Algorithm 3 shows that:

- Running time of $findEnclosingUIEs(e)$ on line 3 depends on concurrency measure γ, i.e., in $O(\gamma)$ time;
- Similarly, the for loop on lines 3-7 is repeated γ times;



(a) Mining Time for Rule Base (ARM) and Pattern Registry (GSP)



(b) Detection Time per Event

Figure 8.5: Mining and Detection Times for ARM and GSP Algorithms

- $findPTSs()$ on line 4 carries out a DFS search in U_i^+. Its runtime depends on the size of U_i^+ which is UIE specific, but can be amortized to $O(\gamma)$ time;

- *patternLookup*() is hash table-based and runs in $O(1)$.

In theory, we can therefore conclude that *measureAnomaly*() in Algorithm 3 runs in $O(\gamma)$ time, that is, proportional to the concurrency measure. After realizing that once a UIE and its associated data structures (such as DAGs for the PTSs) are constructed in memory, they can be cached and reused, I further optimized Algorithm 3 so that the amortized running time per event is nearly $O(1)$, regardless of the system concurrency level.

The average detection times for both methods are shown in Figure 8.5 (b) for comparison. We see that detection time based on the GSP-produced pattern registry is about 30ms and not very sensitive to system concurrency changes, whereas the detection times based on the ARM-produced rule base are much longer and sensitive to concurrency. Note again the logarithmic scale and how drastically the revised framework has improved its performance.

The plot indicates the GSP-based detection algorithm is highly capable of keeping up with high-volume user activities. Detection performance can be further improved by introducing parallel processing (e.g. using elastic computing frameworks such as MapReduce), allowing the framework to scale linearly in accordance with system load.

8.5 Chapter Summary

In this chapter I have made significant enhancements to the ARMOUR framework in terms of accuracy, efficiency, and practicality. They include the following:

- By mining a component-based architecture model using probabilistic, use case-driven sequential patterns (as opposed to association rules without temporal information), I have drastically reduced the framework's space complexity and improved its computational performance.

- The sequential patterns also allowed us to treat network timing errors more rigorously.

- The framework can autonomously determine the optimal training set size and, in doing so, self-determine whether it is in an effective state.

My further evaluations of the framework against the EDS system also validated the following insights:

- ARMOUR is not sensitive to minimum support settings as long as it is at a low level;
- Given the architecture-level positioning of out approach, ARMOUR is effective against potential unknown threats;
- ARMOUR enjoys other practical advantages such as fast, parallelizable detections at near real time and the ability to run unsupervised without clean training data.

In the next chapter we shift our attention from threat detection to threat mitigation, with continued focus on the role software architecture plays in self-protection.

Chapter 9: Mitigating Security Threats with Self-Protecting Architecture Patterns

9.1 ARMOUR Framework Revisited: The Complete Picture

Following the MAPE-K feedback loop for a self-protecting software system, we can see that the mined component interaction model contributes to the *Knowledge* part of the meta-level subsystem and the security threat detection algorithm falls in the *Monitor* step. Given the architecture focus, let's now turn our attention to how software architecture models and adaptation techniques may be used to mitigate security threats at runtime through the *Analysis*, *Planning*, and *Execution* steps. To that end, I will introduce the concept of *architecture-level self-protection patterns* and use several example patterns to illustrate how they provide disciplined and repeatable adaptation strategies for solving well-known security threats. In this chapter I will illustrate their application in dealing with commonly encountered security threats in the realm of web-based applications. Further, I will describe my work in realizing some of these patterns on top of an existing architecture-based adaptation framework, namely Rainbow [65].

The completed architecture of ARMOUR, with the pattern-based threat mitigation added, is depicted in Figure 9.1.

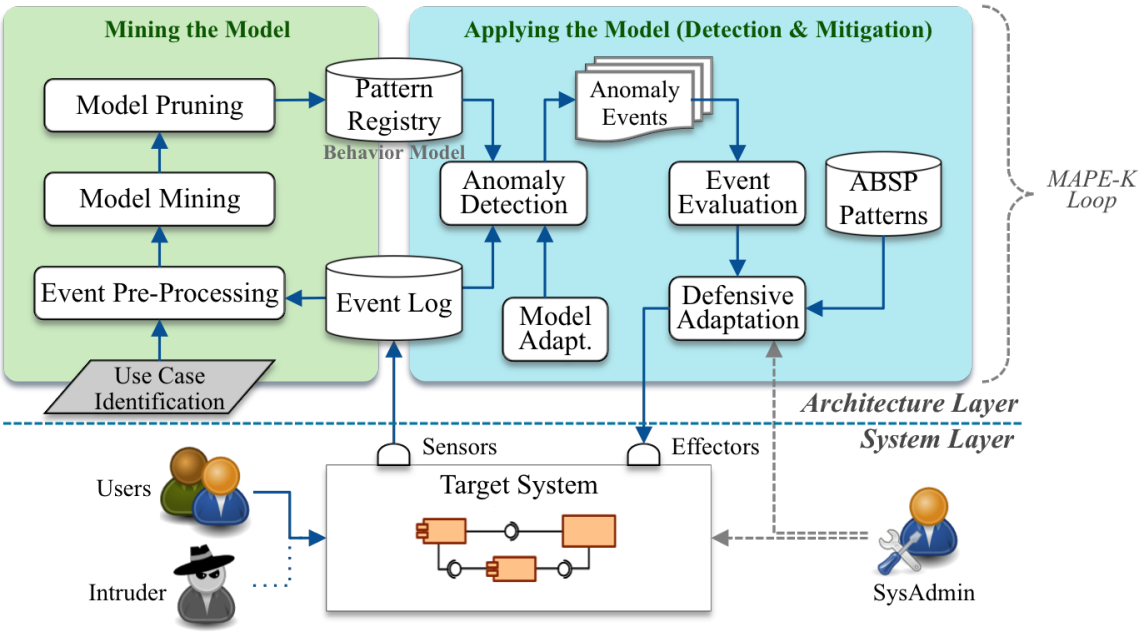


Figure 9.1: ARMOUR Framework: The Complete Picture

9.2 Architecture Patterns for Self-Protection

As mentioned in Chapter 5, repeatable architectural patterns have started to emerge from the software self-protection mechanisms employed in past research efforts. Each pattern has its unique advantages in mitigating certain classes of security threats but each also has its limitations and potential weaknesses. Table 9.1 catalogs the major patterns, illustrates them with examples, and summarizes their pros and cons.

It is worth noting that the self-protection patterns described here represent architectural level adaptation strategies, and are therefore different from previously identified reusable security patterns [70, 199], which constitute for the most part implementation tactics such as authentication, authorization, password synchronization, etc.

Table 9.1: Catalog of Architecture Level Self-Protection Patterns

Pattern Definition	Examples	Evaluation
	Structural Patterns	
Protective Wrapper—Place a security enforcement proxy, wrapper, or container around the protected resource, so that request to / response from the resource may be monitored and sanitized in a manner transparent to the resource.	The SITAR system [187] protects COTS servers by deploying an adaptive proxy server in the front, which detects and reacts to intrusions. Invalid requests trigger reconfiguration of the COTS server. Virtualization techniques are increasingly being used as an effective protective wrapper platform. VASP [207], for example, is a hypervisor-based monitor that provides a trusted execution environment to monitor various malicious behaviors in the OS.	*Pros*– Security adaptation is transparent to the protected resource; easy to implement. *Cons*– Since the wrapper is inherently intended for outside threats, this pattern cannot address security vulnerabilities inside the protected component. The wrapper, esp. when externally visible, may itself become an exploitation target.
Agreement-based Redundancy—In addition to reliability and availability benefits provided by the common redundancy mechanism, this pattern uses agreement-based protocols among the replicas to detect anomalies and ensure correctness of results.	The seminal work by Castro and Liskov [30] described a Byzantine Fault Tolerance (BFT) algorithm that can effectively tolerate f faulty nodes with $3f+1$ replicas. Similar agreement-based voting protocols have been used in many other systems such as SITAR and [179]. The Ripley system [183] implements a special kind of agreement based technique by executing a "known good" replica of a client-side program on the server side.	*Pros*– Robust mechanism that can meet both system integrity and availability goals; effective against unknown attacks. *Cons*– Due to required number of replicas, needs significant hardware and software investments, which can be costly. Further, by compromising enough replicas, the system will be essentially shut down, resulting in denial of service.
Implementation Diversity—Deploy different implementations for the same software specification, in the hope that attacks to one implementation may not affect others. This may be achieved through the use of diverse programming languages, OS, or hardware platforms. To safely switch requests from one instance to another, checkpointing is necessary to save the current system state.	The HACQIT system [148, 149] achieves diversity by using two software components with identical functional specifications (such as a Microsoft IIS web server and an Apache web server) for error detection and failure recovery. Similarly, the DPASA system [40,134] included controlled use of hardware and OS level diversity among redundant environments as part of a comprehensive survivable architecture. A similar approach is dubbed Architecture Hybridization in the MAFTIA system [182].	*Pros*– Effective defense against attacks based on platform-specific vulnerabilities. Increases system resilience since an exploited weakness in one implementation is less likely to kill the entire system. *Cons*– Significant efforts required to develop, test, and deploy diverse program implementations. Diverse languages / platforms may give rise to more software defects. Further, checkpointing to preserve program state at runtime may prove technically challenging.
Countermeasure Broker—The self-protecting system includes a brokering function that, based on the type of an attack, performs dynamic matching or tradeoff analysis to select the most appropriate response or countermeasure, often from a pre-defined repository.	Case-based Reasoning (CBR) techniques are sometimes used to detect intrusions and select responses. The SoSMART system [123] describes an agent-based approach where the CBR "brain" picks the suitable response agent. Alternatively, the ADEPTS effort [60, 194] uses attack graphs to identify possible attack targets and consequently suitable responses.	*Pros*– Flexibility and dynamic nature of the response, when implemented correctly, makes it harder for an adversary to predict and exploit the security defense. *Cons*– Not effective against unknown attacks. Static, "knee-jerk" like responses are likely to be predictable, thus lose effectiveness in the long run. The broker component may become a high value target for adversaries.

Continued on next page

Table 9.1 – continued from previous page

Pattern Definition	Examples	Evaluation
Aspect-Orientation— Following the Aspect Oriented Programming (AOP) principle, this pattern deploys self-protection mechanisms as a separate aspect in the system, transparent to application logic. This pattern is often assisted by Model Driven Engineering (MDE) techniques, as mentioned in Section 5.4.2.	Morin et al. [120], for example, showed how a security access control meta-model is defined and combined with the business architecture meta-model; the security aspect is "woven" into the overall model using MDE tools such as the Kermata language. A related effort [122] shows how the security policies from the model are generated in XACML and integrated into the application using AOP. Xiao et al. [195, 196] also used a similar model driven approach to model security policy rules and dynamically weave them into the runtime application in an agent-based environment.	*Pros*– Bringing AOP benefits to self-protection, such as reuse, separation of concerns, and improved application quality. Assisted with MDE techniques, it also provides a way of expressing security policies as models. *Cons*– It is not yet known if self-protection as a cross-cutting concern can be adequately expressed in today's modeling languages and tools. Weaving in the security aspect into the very application it is protecting makes security logic just as vulnerable.
Behavioral Patterns		
Protective Recomposition—Dynamically adapt security behavior of a system through altering how security-enforcing components are connected and orchestrated. This may include tuning of security parameters, changing authentication / authorization methods, switching to a different crypto algorithm, or regeneration of access control policies.	The E2R Autonomic Security Framework [74, 154], for example, allows nodes in a wireless network to collect and derive security context information from neighboring nodes and reorganize upon node failures. Other examples include changes in contract negotiations between security components [58], altered security service sequences based on QoS objective changes [111], or regenerating new concrete policy instances from a generic policy based on dynamic security context [48].	*Pros*– A key pattern that touches the very essence of self-adaptive security: the ability to adapt security posture based on changing threats and real time security contexts. Makes the system more defendable and harder to exploit. *Cons*– Dynamic and sometimes even non-deterministic security behavior is difficult to test and even harder to evaluate its correctness and effectiveness.
Attack Containment—A simple pattern that seeks to isolate a compromised component from the rest of the system to minimize the damage. Typical techniques include blocking access, denying request, deactivating user logins, and shutting down the system component.	De Palma et al. [47] developed an approach for clustered distributed systems that involves isolating a compromised machine from the network. Solitude uses file-system level isolation and application sand boxing to limit attack propagation [84]. SASI [54], a code-level containment approach, uses a compile-time Software Fault Isolation (SFI) method to enforce security policies.	*Pros*– Simple, fast, and effective way to mitigate and contain security compromises. *Cons*– Often carried out at the opportunity cost of (at least temporary) unavailability of system resources to legitimate users.
Software Rejuvenation—As defined by Huang et al. [80], this pattern involves gracefully terminating an application and immediately restarting it at a clean internal state. Often done proactively and periodically.	In addition to the rejuvenation-based SCIT system [79, 124] and the aforementioned HACQIT system, the Proactive Resilience Wormhole (PRW) effort [161, 162] also employs proactive rejuvenation for intrusion tolerance and high availability. Wang et al. [188] presented a special case of rejuvenation involving software hot-swapping, i.e. swapping out infected components at runtime, replaced with a valid and more strongly protected equivalent.	*Pros*– Effective technique that addresses both security and high availability. Periodic software rejuvenation limits the damage of undetected attacks. *Cons*– The rejuvenation process, short as it may be, temporarily reduces system reliability. Extra care is needed to preserve application state and transition applications to a rejuvenated replica. Extra investment is needed for maintaining redundant replicas.

Continued on next page

Table 9.1 – continued from previous page

Pattern Definition	Examples	Evaluation
Reconfiguration on Reflex— A bio-inspired pattern that reconfigures the system to a higher level of protection (which may be more resource consuming), and when attack passes, returns to a less restrictive mode.	The Security Adaptation Manager (SAM) [77] dynamically operates the system at 3 levels of implementations (calm, nervous, panic) depending on the threat level. The DASAC approach [86] uses a boosting-based algorithm and heuristically defined security levels that allow the agent network to react to agent trustworthiness.	*Pros*– A technique for balancing competing goals of security and performance, depending on the ongoing threat level. *Cons*– Ineffective in the case of undetected threats. An attacker can trick the system to always run at heightened security levels to sacrifice performance, therefore not suitable for persistent threats.
Artificial Immunization— Inspired by adaptive immune systems in vertebrates, this pattern seeks to capture samples of worms or viruses; analyze the virus to derive a signature that can be used to detect and remove it from infected resources; and disseminate the "antidote" to all vulnerable systems.	Kephart et al. [88] and White et al. [192] designed one of the first Digital Immune Systems in response to early virus epidemics, when it was realized that automation was needed to spread the cure faster than the virus itself. Later approaches such as SweetBait [141] use more sophisticated "honeypot" techniques to capture suspicious traffic and generate worm signatures. A variant of this pattern uses the so-called Danger Theory to as the basis for autonomic attack detection and defense [146, 170].	*Pros*– "Detect once, defend anywhere"; the pattern proved to be a successful approach for defending against computer viruses and helped creation of the anti-virus industry. *Cons*– Centralized and top-down architecture is a challenge to scalability and agility, esp. as attacks become more localized and targeted. Further, just like all signature-based techniques, it is not effective against unknown / zero-day attacks.

9.3 ABSP Patterns in Action

In this section, I pick a few patterns from Table 9.1 to illustrate how ABSP may be used to bring self-securing capabilities to a system such as the Znn news service introduced in Chapter 2. For each pattern, I briefly describe the security threat(s) it can be effective for, how the threat could be detected, and finally how it could be dealt with through adaptation.

9.3.1 Protective Wrapper Pattern

Architectural Adaptation

Protective wrappers are not uncommon in past self-protection research. One straightforward way to employ this pattern for Znn is to place a new connector called "Application Guard" in front of the load balancer, as shown in Figure 9.2. To support this change, the Architecture Manager (AM) not only needs to connect to the application guard via the event bus, but also needs to update its architecture model to define additional monitoring events for this new element (e.g. suspicious content alerts) and additional effector mechanisms for its

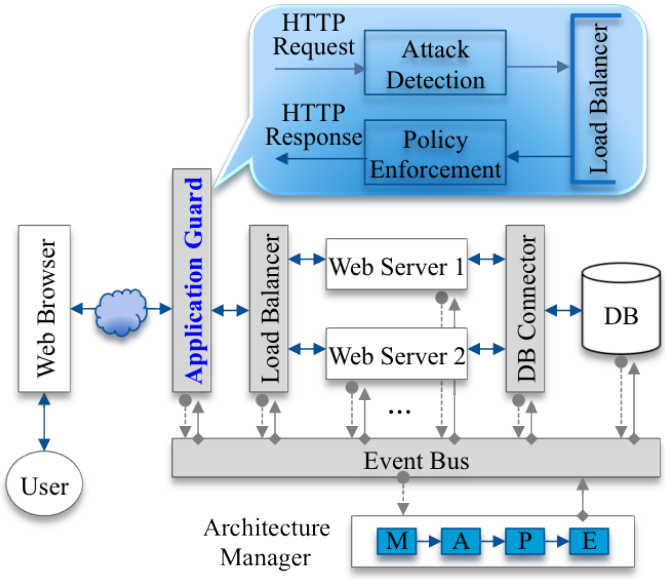


Figure 9.2: Znn Protective Wrapper Architecture

adaptation (e.g. blocking a user).

Threat Detection

The application guard serves as a protective wrapper for the Znn web servers by performing two key functions: attack detection and policy enforcement. By inspecting and if necessary sanitizing the incoming HTTP requests, the application guard can detect and sometimes eliminate the threats before they reach the web servers. Injection attacks (OWASP A1), for example, often contain special characters such as single quotes which will cause erroneous behavior in the backend database when the assembled SQL statement is executed. By performing input validation (e.g. using a "white list" of allowed characters) or using proper escape routines, the wrapper can thwart many injection attempts.

As its name suggests, this protective wrapper works at the application level, in contrast to conventional network firewalls that focus on TCP/IP traffic. It communicates with and is controlled by the model-driven Architecture Manager "brain", and as such can help detect more sophisticated attack sequences that are multi-step and cross-component. For

example, the AM can signal the application guard to flag and stop all access requests to the web server document root, because a sensor detected a buffer overflow event from the system log of the web server host. The latter may have compromised the web server and placed illegitimate information at the document root (e.g., a defaced homepage, or confidential user information). The detection may be achieved through incorporating an attack graph in the AM's architecture model, as described in [60].

Threat Mitigation

A second function performed by the application guard is policy enforcement as directed by the AM. Take Broken Authentication and Session Management (OWASP A3) for example; web sites often use URL rewriting which puts session IDs in the URL:

```
http://znn.com/premium/newsitem?sessionid=SIRU3847YN9W38475N&
newsid=43538
```

When this URL is either shared or stolen, others will be able to hijack the session ID to access this user's content or even his/her personal information. The application guard can easily prevent this by applying encryption / obfuscation techniques so session IDs cannot be identified from the URL, or by tying session IDs with user's MAC addresses so that session IDs cannot be reused even if they are stolen. Similar mechanisms at the application guard may also help patch up other vulnerabilities such as Insecure Direct Object References (OWASP A4) and Failure to Restrict URL Access (OWASP A8).

More adaptive enforcement actions are possible thanks to the AM component that is aware of the overall system security posture. After the AM senses the system is under attack, it can instruct the application guard to dynamically cut off access to a compromised server, switch to a stronger encryption method, or adjust trustworthiness of certain users. Adaptation strategies may be based on heuristic metrics indicating the overall system's security posture, which are computed in real time. As a concrete example, in section 9.4 I will show how this pattern is employed against denial of service (DoS) attacks.

9.3.2 Software Rejuvenation Pattern

As described in Table 9.1, the Software Rejuvenation pattern involves gracefully terminating an application instance and restarting it at a clean internal state. This pattern is part of a growing trend of proactive security strategies that have gained ground in recent years. By proactively "reviving" the system to its "known good" state, one can limit the damage of undetected attacks, though at the cost of extra hardware resources.

Architectural Adaptation

When applying this pattern to the Znn application, I will update the AM's system representation to establish two logical pools of web servers: in addition to the active server pool connected to the load balancer, there will also be an idle web server pool containing running web servers in their pristine state, as shown in Figure 9.3. These server instances could be either separate software processes or virtual machine instances. In the simplest case, the AM will issue rejuvenation commands at regular intervals (e.g., every 5 minutes, triggered by a timer event), which activate a new web server instance from the idle pool and connects it to the load-balancer. At the same time, an instance from the active pool will stop receiving new user requests and terminate gracefully after all its current user sessions log out or time out. The instance will then be restarted and returned to the idle pool. The process is illustrated in Figure 9.4.

The AM "brain" may also pursue more complex rejuvenation strategies, such as:

- Use threat levels or other architectural properties to determine and adjust rejuvenation intervals at runtime
- Perform dynamic reconfigurations and optimizations (e.g. restart a server instance with more memory based on recent server load metrics)
- Mix diverse web server implementations (e.g. Apache and Microsoft IIS) to thwart attacks exploiting platform-specific vulnerabilities

Note that, the rejuvenation process, short as it may be, temporarily reduces system reliability. Extra care is needed to preserve application state and transition applications to a

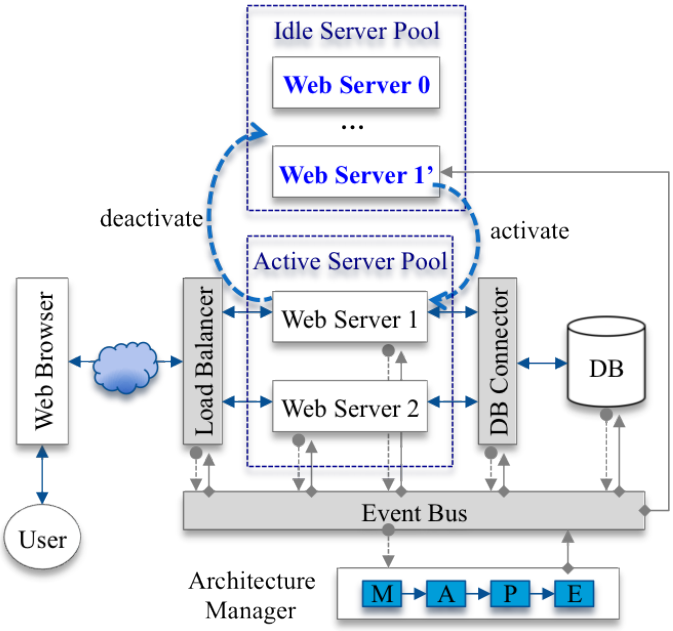


Figure 9.3: Znn Software Rejuvenation Architecture

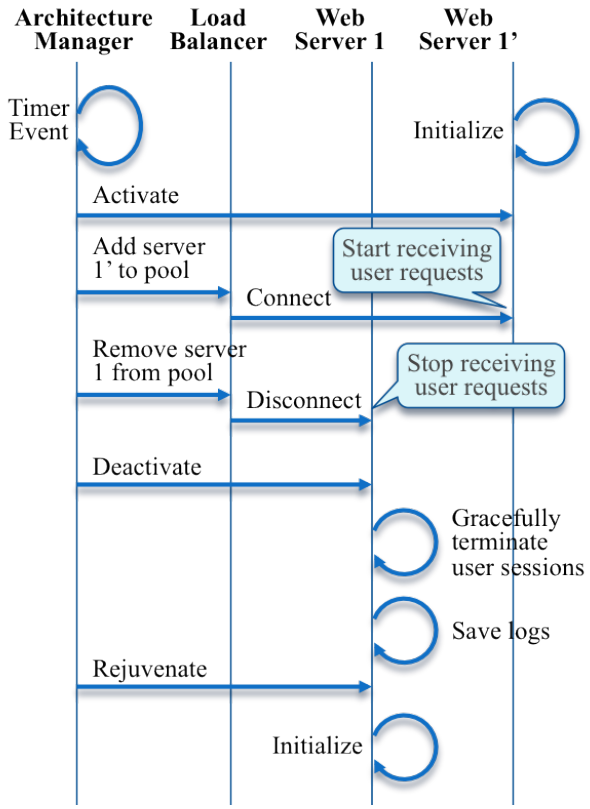


Figure 9.4: Web Server Rejuvenation Process

rejuvenated replica.

Threat Detection

A unique characteristic about the rejuvenation pattern is that it neither helps nor is dependent upon threat detection, but for the most part used as a mitigation technique.

Threat Mitigation

Although the rejuvenation pattern doesn't eradicate the underlying vulnerability, it can effectively limit potential damage and restore system integrity under injection (OWASP A1), reflective XSS (OWASP A2), and to some extent CSRF (OWASP A5) attacks, detected or undetected. These are considered among the most vicious and rampant of web application threats, in part because the attack vector is often assisted by careless or unsuspecting users. Clicking on a phishing URL is just one of the many examples. When a fragment of malicious code is sent to the server, such as the following that steals user cookies [132]:

```
<SCRIPT type="text/javascript">
var adr = 'example.com/evil.php?cakemonster='
          + escape (document.cookie);
</SCRIPT>
```

This piece of injected code may be stored in server memory or (in a worse case) in the database, and then used for malicious intents such as stealing confidential user information, hijacking the server to serve up malware, or even defacing the website - and continue doing so *as long as the server is running.*

With a rejuvenation pattern in place, a server may only be compromised for up to the rejuvenation interval. In mission-critical operations, the interval can be as short as a few seconds, drastically reducing the probability and potential damage from these attacks even when detection sensors fail.

Our pattern implementation as depicted in Figure 9.3 does have some limitations. First, for persistent attacks such as DoS, rejuvenating the web server will not be effective because

the DoS traffic will simply be directed to the new web server instance and overwhelm it. In such cases rejuvenation must be carried out in conjunction with other countermeasures such as blocking the attacking source. Secondly, caution must be taken so that corrupted state is not migrated to the new instances. For example, when malicious code is stored in the database, simply recycling the web server will not eradicate the root of the threat because restarting a database server instance will only clean up transient, in-memory storage but has no effect on data changes already committed to permanent storage. This pattern, therefore, is not an effective mechanism against stored XSS attacks.

9.3.3 Agreement-based Redundancy

As pointed out in the previous subsection, proactively "hot swapping" active and possibly tainted web servers with new pristine instances can effectively limit the damage of scripting attacks that seek to inject malicious code in the web server, but the technique can offer little relief to attacks that have succeeded in permanently altering the system state, particularly in the database. To address the latter challenge, I consider another architecture pattern, Agreement-based Redundancy, which maintains multiple replica of a software component at runtime in order to tolerate Byzantine faults and ensure correctness of results. A prime example of this pattern comes from the seminal work by [30] described a Byzantine Fault Tolerance (BFT) algorithm that can effectively tolerate f faulty nodes with $3f+1$ replicas within a short window of vulnerability. The strengths of this pattern is many-fold - it is easy to implement, performs well, helps meet both system security and availability goals, and is effective against unknown attacks.

Architectural Adaptation

In the Znn example I choose to apply this pattern to the database layer, as shown in Figure 9.5. First, I update the AM's architecture representation to maintain a number of identical database instances (along with their respective database connectors), all active and running concurrently. Secondly, a new connector called DB Guard is introduced to handle database

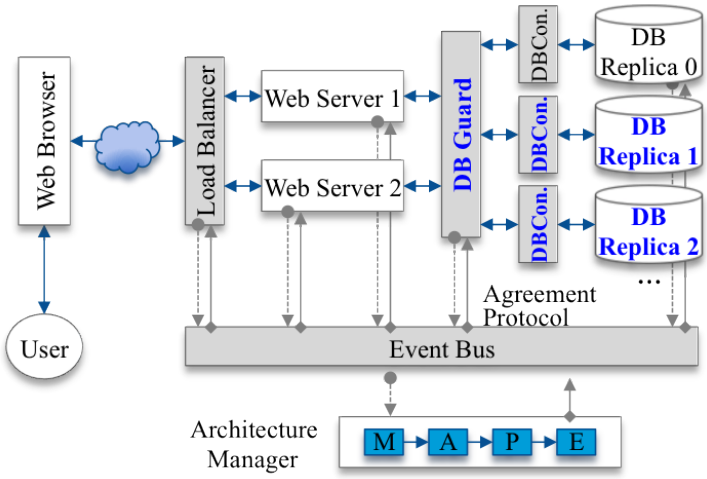


Figure 9.5: Znn Agreement-Based Redundancy Architecture

requests from web servers using an agreement-based protocol. The AM communicates the agreement-based protocol specifics to the DB Guard, such as the number of replicas and quorum thresholds. The AM can dynamically adapt the protocol as needed at runtime.

Threat Detection

Given the heavy reliance on databases in today's software applications, it is no surprise SQL injection is ranked as the number one threat in both OWASP Top 10 and CWE Top 25. The Architecture-based Redundancy pattern can effectively detect and stop the SQL variants of the injection attack (OWASP A1) and stored XSS (OWASP A2) attack when they contain illegal writes to the Znn database. Consider a simplified scenario where the Znn web server attempts to retrieve news articles from a database table based on keyword:

```
...
string kw = request.getParameter("keyword");
string query = "SELECT * FROM my_news
  WHERE keyword = '" + kw + "'";
...
```

Note that many database servers allow multiple SQL statements separated by semicolons to be executed together. If the attacker enters the following string:

```
xyz'; DELETE FROM news; --
```

Then two valid SQL statements will be created (note the training pair of hyphens will result in the trailing single quote being treated as a comment thus avoiding generating a syntax error, see [176] for details):

```
SELECT * FROM my_news WHERE keyword='xyz';
DELETE FROM news;
```

As a result, a seemingly harmless database query could be used to corrupt the database and result in loss of data. Good design and coding techniques, along with static analysis tools can help identify vulnerabilities. As mentioned earlier in the paper, however, such efforts are labor-intensive and not bullet-proof. Using the Agreement-based Redundancy pattern, I take an non-intrusive approach that does not require code changes to the web application nor the database. Instead, I execute the following algorithm in the DB Guard connector:

1. Treat each database request R as a potential fault-inducing operation, and execute it first on the primary node (replica 0). The selection of the primary node is arbitrary.

2. Use a voting algorithm to check predefined properties of the database. For example, one property may be the number of news articles. The AM is responsible for defining and monitoring these properties and making them available to the database connector. When quorums can be reached on all properties and the primary node is part of the quorum, proceed to next step; otherwise flag R as invalid and revert the primary node to its state before R, either by rolling back the transaction or by making a copy of another replica.

3. Execute R on all other replicas 1 to n, bringing all replicas to the same state.

4. Adjudicate the results from all replicas using the voting algorithm. If a quorum is reached, return the result to client; otherwise consider the system in a compromised state and raise flag for human administrator attention.

Note that the last step of the algorithm is added to take into account other network- and host-level attacks that may have compromised a database replica through "back doors".

It is easy to see that when the above attack string gets sent to the DB Guard and executed in the primary node, the number of news articles is reduced to zero after the delete command, therefore different from the quorum. The request will be aborted and the system reverted to its valid state. My implementation, however, comes with a caveat: it is not effective when the SQL injection seeks only to read data (i.e. the compromise is in system confidentiality, not integrity). In the latter case, the protective wrapper pattern can still help inspect and detect the anomaly.

Threat Mitigation

As we have seen from the above scenario, the SQL injection attack is effectively stopped after it is detected in the algorithm. To complete the full sequence for threat mitigation, I only need to furbish a few more details:

- Once an invalid and potentially malicious request is detected, the DB Guard will notify the AM that can deploy countermeasures such as nullifying the associated user session, notifying the system administrator, or even disabling the user account in question.

- When the adjudication of the results (step 4 of the algorithm) is not unanimous, raise a flag about the minority server instance. If further diagnostics confirm the instance is not in a valid state, destroy and regenerate this instance.

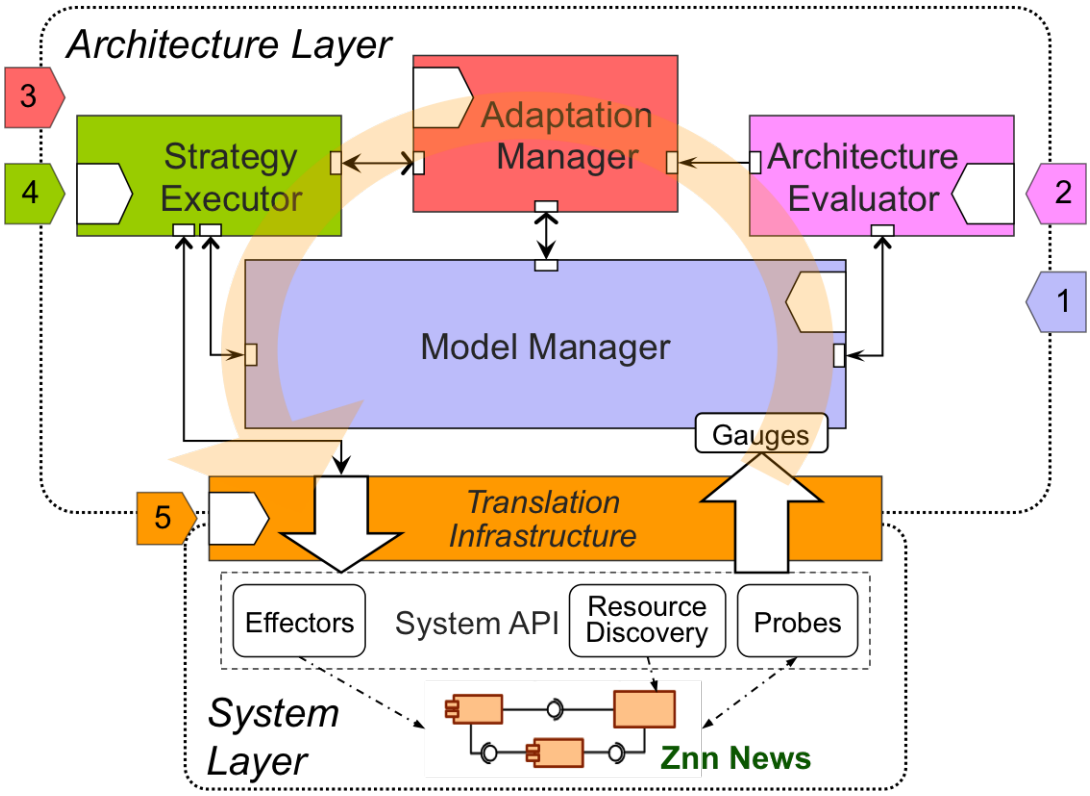


Figure 9.6: Rainbow Framework Architecture

9.4 Realizing Self-Protection Patterns in Rainbow

In this section, I outline how to implement this approach in an architecture-based self-adaptive framework called Rainbow. I begin by providing an overview of Rainbow, and then continue by discussing how the Protective Wrapper pattern can be realized by the framework.

9.4.1 Rainbow Framework Overview

The Rainbow framework has demonstrated how architecture models of the system, updated at runtime, can form the basis for effective and scalable problem detection and correction. Architecture models represent a system in terms of its high level components and their interactions (e.g., clients, servers, data stores, etc.), thereby reducing the complexity of

those models, and providing systemic views on their structure and behavior (e.g., performance, availability, protocols of interaction, etc.). In the context of this paper, the Rainbow framework can be viewed as Architecture Manager capable of evaluating and adapting the underlying system to defend against threats.

The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. Figure 9.6 shows the adaptation control loop of Rainbow. Probes are used to extract information from the target system that update the architecture model via Gauges, which abstract and aggregate low-level information to detect architecture-relevant events and properties. The architecture evaluator checks for satisfaction of constraints in the model and triggers adaptation if any violation is found, i.e., an adaptation condition is satisfied. The adaptation manager, on receiving the adaptation trigger, chooses the "best" strategy to execute, and passes it on to the strategy executor, which executes the strategy on the target system via effectors.

The best strategy is chosen on the basis of stakeholder utility preferences and the current state of the system, as reflected in the architecture model. The underlying decision making model is based on decision theory and utility [127]; varying the utility preferences allows the adaptation engineer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [35], is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, namely a tactic, on the target system with variable execution time. A tactic defines an action, packaged as a sequence of commands (operators). It specifies conditions of applicability, expected effect and cost-benefit attributes to relate its impact on the quality dimensions. Operators are basic commands provided by the target system.

As a framework, Rainbow can be customized to support self-adaptation for a wide variety of system types. Customization points are indicated by the cut-outs on the side of the architecture layer in Figure 9.6. Different architectures (and architecture styles), strategies, utilities, operators, and constraints on the system may all be changed to make Rainbow reusable in a variety of situations. In addition to providing an engineering basis

for creating self-adapting systems, Rainbow also provides a basis for their analysis. By separating concerns, and formalizing the basis for adaptive actions, it is possible to reason about fault detection, diagnosis, and repair. For example, many of the standard metrics associated with classical control systems can, in principle, be carried over: settling time, convergence, overshoot, etc. In addition, the focus on utility as a basis for repair selection provides a formal platform for principled understanding of the effects of repair strategies.

In summary, Rainbow uses architectural models of a software system as the basis for reasoning about whether the system is operating within an acceptable envelope. If this is not the case, Rainbow chooses appropriate adaptation strategies to return the system to an acceptable operating range. The key concepts of the approach are thus: (a) the use of abstract architectural models representing the run-time structures of a system, that make reasoning about system-wide properties tractable, (b) detection mechanisms that identify the existence and source of problems at an architectural level, (c) a strategy definition language called Stitch that allows architects to define adaptations that can be applied to a system at runtime, and (d) a means to choose appropriate strategies to fix problems, taking into consideration multiple quality concerns to achieve an optimal balance among all desired properties.

9.4.2 Realizing the Protective Wrapper Pattern

Now I proceed to describe how the Protective Wrapper Pattern in Section 9.3.1 is implemented in Rainbow to protect Znn against a denial of service (DoS) attack. DoS has been extensively researched in the past, including recent efforts using adaptive approaches [15,105]. My focus in this chapter is not so much on advancing the state of the art for DoS attack mitigation, but on illustrating how the problem may be addressed at the architectural level using repeatable patterns and a runtime self-adaptive framework.

Architecture Adaptation

At the system level, the protective wrapper is placed in front of the load balancer of Znn to achieve two levels of protection: 1) it maintains a list of black-listed IPs that are considered to be malicious, and ignores all requests from them; and 2) it maintains a list of suspicious clients that are generating an unusually large amount of traffic and limits the requests that get forwarded to the load balancer. Each of these are manipulated via effectors in Znn (in reality, scripts that can run on the load balancer) that introduce and configure the wrapper. Each script is associated with an architectural operator that can be used by tactics in Stitch to implement the mitigation.

The architecture model of Znn is annotated with properties to determine the request rates of clients, and the probability that a client is engaging a DoS (i.e., being malicious). Gauges report values for these properties (described below), and constraints check that clients have reasonable request rates and low probabilities of maliciousness, and if not, are throttled or on the blacklist. If these constraints fail, then the mitigation strategy above is applied.

In terms of customization of Rainbow, the model and its annotation with the above properties corresponds to customization point 1 in Figure 9.6, and the constraints that check the correctness of the model to point 2.

Threat Detection

The DoS attack is detected by probes that monitor the traffic in the system, and correspond partially to customization point 5 in Figure 9.6. Rainbow aggregates this data into actionable information within gauges, and then uses this information to update the architectural model to reflect operating conditions. To determine the probability of a client participating in a DoS, we follow the approach described in [27, 28]. We define transactions representing request behaviors in the architectures that are derived from low level system events, and an Oracle that analyzes the transactions from each client and, using a method called Spectrum-based Fault Multiple Fault Localization, reports the probability of each client

acting suspiciously as a *maliciousness* property on each component in the model.

Furthermore, probes and gauges keep track of which clients are in the blacklist or being throttled, allowing the constraints in the model to fail only on clients that haven't been dealt with yet.

Threat Mitigation

When a threat is detected and reported in the architectural model, causing a constraint to fail, the Rainbow Adaptation Manager is invoked. It selects and executes adaptations to maximize the utility of the system. In the case of a DoS attack, maximizing utility means stopping the attack with minimal client service disruption. That is, the DoS response must take care not deny access to clients without cause.

Consider the scenario where attackers may be dealt with differently depending on the frequency with which they attack (i.e. repeat offenders) and the duration of the attack. Rules to determine how these factors influence response could be encoded into a security policy with the following logic:

- The traffic for previously unknown attackers is throttled (i.e., some requests are ignored) to limit the impact of the attack without totally cutting off service. This approach minimizes the chances of disrupting the service of possibly legitimate clients.
- Repeat offenders are *blackholed* meaning all that client's traffic is filtered at the load balancer. Known malicious clients are given less mercy than those who have not previously attacked Znn.
- Long-running attacks are not tolerated under any circumstances.

These rules are encoded as the Rainbow strategy shown in Listing 9.1. The applicability of the FixDosAttack strategy depends on the state of the system as it is represented in the architectural model. The strategy is only applicable if the cUnderAttack condition is true in

the model.[1] Conditions such as cLongAttack (line 3) and cFreqAttacker (line 9) reflect the state of architectural properties, such as whether an attack is ongoing, and act as guards in the strategy's decision tree. This strategy captures the scenarios where infrequent and frequent attackers are dealt with by throttling or blackholing the attack respectively, unless the attack is long running. This is consistent with the logic described above.

Listing 9.1: An example strategy for implementing the DoS Wrapper.

```
1  strategy FixDoSAttack [cUnderAttack] {
2    t0: (cInfreqAttacker) -> throttle() @[2000(/*ms*/ ] {
3      t0a: (cLongAttack) ->  blackhole () @[2000] {
4        t0ai: (default) -> done;
5      }
6      // No more steps to take
7      t0a: (default) -> TNULL;
8    }
9    t1: (cFreqAttacker) -> blackhole() @[2000/*ms*/] {
10     t1a: (cLongAttack) -> blackhole() @[2000] {
11       t1ai: (default) -> done;
12     }
13   }
14 }
```

While strategies determine which action to take, tactics are responsible for taking the action. If a condition is true in the strategy, then the subsequent tactic is applicable. For example, an infrequent attacker would cause the cInfreqAttacker condition to be true invoking the throttle() tactic (line 2). Tactics are the specific actions to take to transform the architecture into a desired state. The tactics used in the FixDoSAttack strategy are: throttle and blackhole.

Consider the blackhole tactic shown in Listing 9.2. When executed, this tactic will change the ZNN system to discard traffic from specified clients (i.e., put them in a blackhole). The tactic has three main parts: the applicability condition that determines whether the tactic

[1]The details of this condition are elided for space, but are written in the first-order predicate langauge of Acme [66]

is valid for the situation, the tactics action on the architecture, and the tactic's anticipated affect on the model of the system.

Listing 9.2: Tactic to black hole an attacker.

```
1  tactic blackholeAttacker () {
2   condition {
3     // check any malicious clients
4     cUnblackholedMaliciousClients
5   }
6   action {
7    // Collect all attackers
8    set evilClients =
9      { select c : T.ClientT in M.components |
10       c.maliciousness > M.MAX_MALICIOUS};
11   for (T.ClientT target : evilClients) {
12     // black hole the malicious attacker
13     Sys.blackhole(M.lbproxy, target);
14   }
15  }
16  effect {
17   // all the malicious clients blackholed.
18   !cUnblackholedMaliciousClients
19  }
20 }
```

The tactic's applicability condition relies on whether or not a client is currently attacking, indicated by a maliciousness threshold property in the architecture. If a gauge sets the maliciousness property of the suspected attacker above the maliciousness threshold, then blackholing is a viable action to take. In this sense, the tactic provides an additional checkpoint that more closely aligns with the architecture.

The action part of the tactic places clients that are identified as attackers in the blackhole by invoking the Sys.blackhole(...) operation. This operation is bound to an effector that actually modifies the Znn system to drop attacker packets, by adding the client to the

blacklist, adding the filter, and restarting the load balancer.

Finally, the effect section verifies that the action had the intended impact on the system. In terms of acting as a protective wrapper, this tactic adapts Rainbow that it can intercept and discard malicious traffic before it reaches the Znn infrastructure.

The running of the Rainbow software is shown in the screenshot captured in Figure 9.7.

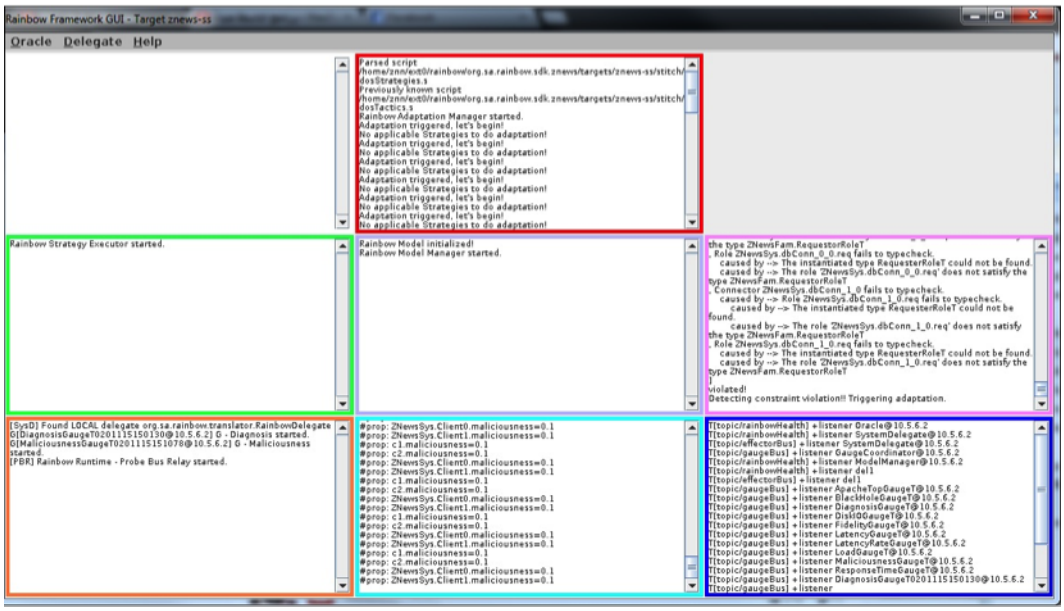


Figure 9.7: Rainbow Framework Screenshot

9.5 Chapter Summary

In this chapter I have showed how architecture-based self-protection patterns help enhance the overall security of a self-adaptive system in ways that traditional, perimeter-based defense mechanisms couldn't. My initial implementation of the Protective Wrapper pattern using the Rainbow framework, though primitive, has demonstrated the feasibility of the ABSP approach and serves as a validation of my Hypothesis #3 from Section 3.2.

Chapter 10: Beyond Self-Protection: Self-Optimization of Deployment Topologies

The overarching strategy of the ARMOUR framework has been mining an architectural-level system behavior model at runtime, and using it as the basis for detecting and mitigating security anomalies. It is foreseeable that the same behavior model may be used to improve other quality attributes of a software system that are not security related.

As a final illustration, I outline an application of my approach for improving the performance of software through adaptive redeployment of its components. Such capability would typically be realized in the Planning phase of the MAPE cycle in a self-optimizing software system.

10.1 Background

The design and development of large-scale, component-based software systems today are influenced by modern software engineering practices as embodied by architecture styles (e.g., pipe and filter), design patterns (e.g. proxies), and coding paradigms (e.g. aspect-orientation). A direct consequence is that the deployment of such systems becomes more complex and fluid, with hundreds or perhaps even thousands of options and parameters to consider, along dimensions such as location, capacity, timing, sequencing, service levels, security, etc. Many of them may be interdependent and possibly conflicting. Due to the combinatorially large problem space, the values of these parameters are usually set and fine-tuned manually by experts, based on rules of thumb and experience.

An objective for autonomic systems is therefore to intelligently navigate the solution space and seek ways to optimally (re)deploy the system to improve the system performance and cost [87].

To illustrate the self-optimization challenge, let's turn our attention to the deployment topology of the EDS system. As a geographically distributed system, some of the components, such as HQUI (recall Figure 6.1), need to reside at the headquarters (HQ) facility, while some, such as the Resource Monitor, are required to be at a remote site to be collocated with emergency response equipment. Other components are more flexible and can be deployed at either locations. Depending on the topology, inter-component messages can be either local (via inter-process communication on a single computer or over a LAN), or remote over a WAN, with the latter having a much larger network latency. Take the strategy analysis use case outlined in Figure 6.2 (a) for example, if the *Strategy Analyzer* and *Strategy Analyzer KB* components reside at different sites, event e_8 may take a much longer time than what it would be if the two components were collocated, adversely affecting the response time experienced by the end user. Obviously, the system should employ a deployment topology that minimizes remote transactions to reduce overall network latency, subject to other constraints.

It is worth noting that this is by no means a new problem, and has been manifested in various settings such as system resource management [138], cloud performance optimization [26], wireless network configuration [114], etc. However, traditional approaches, including our own prior work [112, 114], assume the availability of a detailed component interaction model, that includes information about the component dependencies, frequency of interactions among the components, size of exchanged data, processing sequences, etc. As pointed out earlier in the paper, such a model is difficult to come by and costly to maintain.

My proposed approach, on the other hand, leverages the same component interaction model for dynamic optimization of the deployment topology at runtime, a model that needs no prior development and can stay up to date even when the system behavior shifts. Many modern middleware frameworks that support adaptation (such as [112]) provide facility for redeploying components across distributed locations in a software system. Thanks to new advances in computing infrastructures such as virtualization and cloud computing,

dynamically re-deploying software components is being made easier than ever. Puppet software [143] and CloudFormation service from Amazon Web Services [8], for example, provide programmable mechanisms to create, configure and manage virtualized computing resources on-the-fly.

10.2 Applying Association Rules Mining

It is easy to see that the problem of determining deployment topology, namely, assigning component c_i to location S_j, can be framed as a clustering problem. Intuitively, events that have a higher probability of occurring together should be local (i.e., in the same cluster). One straightforward implementation apporach is to use an *agglomerative hierachical clustering* algorithm [172]: we start with individual components as single-point clusters, then successively merge the two closest clusters until only the desired number of clusters remain. Typically, the "closeness" between two clusters is based on a proximity / distance measure such as the Euclidean distance. In our problem context, we could of course simply observe and compute the average frequency of pair-wise events between two component clusters as the proximity measure. This approach is effective in grouping frequently interacting components together, but has known limitations such as the tendency to reach local optima due to the lack of a global objective—my evaluation will later confirm this.

Here it is obvious that the clustering problem only concerns the proximity of the components and not any temporal information in event executions, therefore I choose ARM as the mining technique. Because our associations mining algorithm produces a rule base that captures not only the frequency count of single pairwise events [1], but also the probability of the *co-occurrence* of a set of component interactions, we can use it to define a better proximity measure. In this application scenario, we are only interested the support value *supp* of an itemset $X \cup Y$ as defined in Section 6.3, which is readily available as an intermediate result from the Apriori algorithm. I now also denote e_{ij} as an event that is initiated from

[1] Note that a single event can be viewed as a EAR of the form $X \rightarrow Y$ where X is the empty set ϕ and $Y = \{e\}$)

component i to component j or vice versa (i.e., $i = e.c_{src}$ and $j = e.c_{dst}$, *or* $j = e.c_{src}$ and $i = e.c_{dst}$).

I now introduce the ***cohesion*** measure of a component cluster C as:

$$Cohesion(C) = \left(\sum_{X \in 2^E} s(X) \right) / |C|$$

where $E = \{e_{ij} \mid \forall i \in C \wedge j \in C\}$ is the set of "local" events within cluster C, and 2^E is the powerset of E. In other words, the cohesion measure of a cluster of components is the sum of the support values of all subsets of local events within C, normalized by the cluster size $|C|$.

Using the cohesion measure, the proximity between two clusters C_1 and C_2 is therefore defined as the ***cohesion gain*** resulting from their would-be merge:

$$proximity(C_1, C_2) = Cohesion(C_1 \cup C_2) - Cohesion(C_1) - Cohesion(C_2)$$

Intuitively, this measure encourages the merge of two sets of components if the merge results in more localized event sequences. We can see that the component interaction model captured from assocations mining can be used to provide a *probabilistic proximity measure* that is informed by system-wide transactions rather than simple pairwise events.

10.3 Evaluation

To evaluate the effectiveness of applying the EARs in improving EDS deployment topology, I instrumented the server-side Java code to simulate network latency based on a configurable topology "metamodel". Before making an inter-component method call, each calling component (StrategyAnalyzer, Repository, etc.) queries the metamodel to determine if it is a local (LAN) or remote (WAN) call and generates a Gaussian-distributed network latency time accordingly.

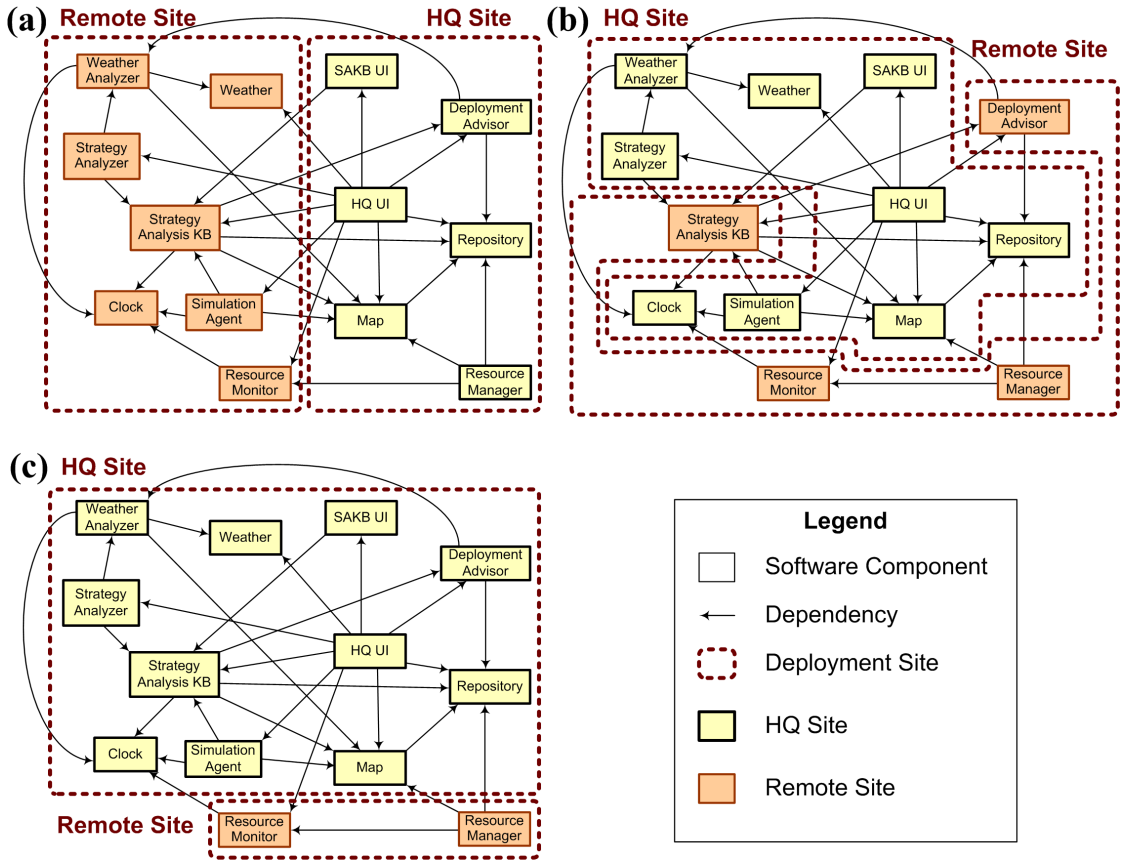


Figure 10.1: Alternative EDS deployments: (a) initial deployment, (b) optimized deployment based on pair-wise proximity clustering technique, and (c) optimized deployment based on the mined component interaction model

In test runs with 20 active concurrent users, I set the mean network latency for LAN and WAN at 10 ms and 100 ms, respectively. The first test run of the system used an arbitrary deployment topology, as shown in Figure 10.1 (a). Here, no special attention was given to the locality of the components: they were divided up more or less evenly between the HQ site and the remote site. System execution logs showed that average event latency is about 44 ms.

As an alternative technique for comparisons sake, I first used a basic hierarchical clustering algorithm that used pairwise event frequency $\sigma(e_{ij})$ between any two components i and j as the distance function. The algorithm recommended a different topology as shown in

Figure 10.1 (b). The second set of test runs conducted under this new topology showed that average event latency was reduced to about 30 ms, indicating a significant improvement.

Now I turn to evaluate my proposed approach based on the component interaction model. After mining the system execution traces, the Apriori algorithm created a rule base that doubled as a probabilistic proximity matrix for any two sets of components. Feeding the cohesion gain based function introduced earlier to the hierachical clustering algorithm, a third topology emerged, as shown in Figure 10.1 (c). Test runs based on the new topology showed that average event latency was further reduced to 20 ms, a 33% reduction compared with the pairwise event frequency based clustering and a 54% reduction compared with the original topology.

I conducted 20 batch runs for each deployment topology, eliminating transients by taking observations only after the system entered a steady state. The average latency for each deployment topology, along with the corresponding 95% confidence interval are shown in Figure 10.2. During the clustering process, I ensured that the $HQUI$ and $ResourceMonitor$ components are pre-assigned to the HQ site and the remote site, respectively (otherwise the optimization would result in all components being assigned to a single site). Statistical tests can also confirm that the latency improvements are not trivial. I have validated that taking into account system-wide event co-occurrences can help overcome the local optima during the clustering process, resulting in improved system performance.

Note that in practice, optimal deployment of resources depends on many other factors besides network latency, such as cost of component re-deployment, hardware capacity at each location, etc. A more holistic approach needs to formulate a higher-level objective function that weighs benefits against various costs and constraints (e.g., as developed in [21] and [112]). In that case, the component-wise probabilistic proximity measure from our model can become an input to the larger optimization algorithm.

The advantage of the architecture-based approach is evident in this application scenario: Even though the mapping from software components to hosts and sites is changed as the result of automated redeployment, the self-adaptation happens within the network and

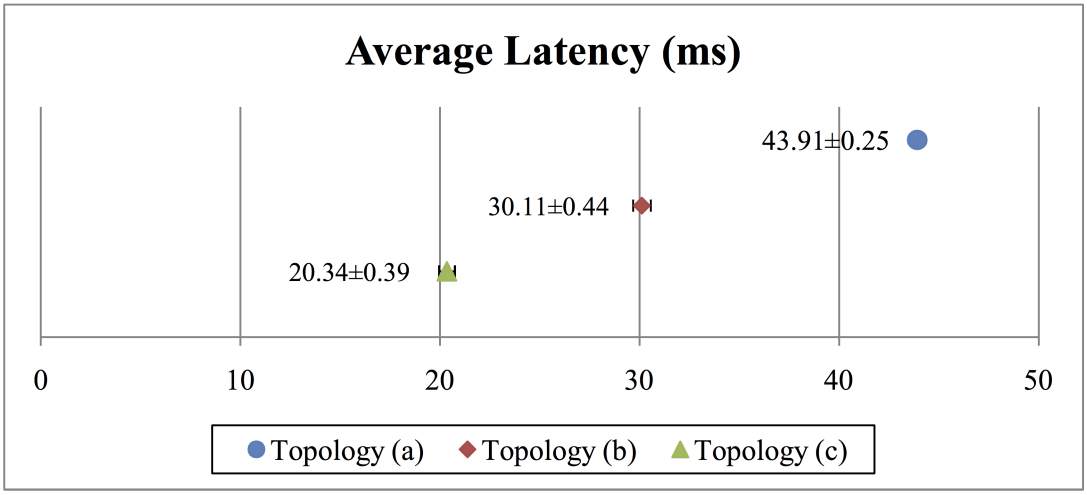


Figure 10.2: System latency for three deployment topologies (average latency at 95% confidence interval)

host configurations of the base-level subsystem (e.g., new URLs for Java RMI calls). The abstract component interaction model within the meta-level subsystem, in the form of EARs captured in the rule base, is transparent to the deployment topology change, therefore remains intact and does not need to be re-learned.

Chapter 11: Related Work

11.1 Related Surveys

Because self-protection mechanisms fall into the intersection of self-adaptive systems and software security, I have sought survey papers from both research domains.

First, even though the research field of self-adaptive and self-managing systems is a fertile research ground with rapidly advancing state of the art [34, 104], little endeavor has been devoted to security as an adaptation property. Nonetheless, a number of related surveys are worth noting. Villegas et al. [184] developed a control theory-based framework for surveying and evaluating self-adaptive software systems, in which security is included as one of the observable adaptation properties. None of their surveyed papers, however, covered security. A taxonomy of compositional adaptation [117] focuses on composition as a key paradigm for adaptation, and describes a taxonomy based on how, when, and where software composition takes place. A related survey can be found in [150] with a stronger focus on adaptive middleware. Even though these two surveys are not directly related to self-protection systems, my research draws certain taxonomy attributes for my purposes. Salehie and Tahvildari [152] presented a comprehensive survey on self-adaptive software in general. It offers a taxonomy of self-adaptation that covers a variety of dimensions, some of which are security-relevant such as adaptation layers (OS, middleware, etc.), realization approach (such as static vs. dynamic decision making), and temporal characteristics (such as reactive vs. proactive adaptation). Even though many of these dimensions are relevant for self-protection, they need to be further defined in the specific context of security before they become useful. A comprehensive survey of self-healing systems [142] provides a taxonomy of system failure classes (security being one of them) and catalogs self-healing approaches such as architecture-based, agent-based, middleware-based, etc. Albeit not security-focused, the

paper identified approaches and techniques that overlap with the self-protection research especially around the security goal of availability, as will be seen later in this paper. Across these surveys, the profound influence of the IBM Autonomic Computing (AC) vision as presented in [87] is clearly visible, specifically around the adopted definitions of self-* properties and the MAPE-K loop. A recent survey, for instance, further expanded the MAPE-K concept with a "Degree of Autonomicity" dimension with four progressive levels of maturity: Support, Core, Autonomous, and Autonomic [81].

Secondly, I have found quite a number of relevant surveys in the software security domain. I recognize that computer security is a vast research domain, and that my objective is not to advance the state of the art of security techniques but rather to apply them to self-protecting systems. Consequently, I have limited my search to high-level surveys and review papers from which I can draw useful attributes for the self-protection taxonomy (described in Section 5.3). To that end, I have found good sources that cover various security concepts:

- To have a better understanding of computer security threats and vulnerabilities, I turned to [82], which provides a state-of-the-art "taxonomy of taxonomies" on types of attacks (general attacks, intrusion detection system (IDS) signatures and anomalies, Denial of Service (DoS) related attacks, web attacks and other specialized taxonomies) and vulnerabilities (software flaws, network vulnerabilities). Similarly, Swiderski and Snyder [169] presented Microsoft's threat model which classifies attacks along the STRIDE model (spoofing, tampering, repudiation, information disclosure, DoS, and elevation of privilege). A different attack taxonomy was introduced in [19], which defined high-level categories, including Disclosure, Modification, DoS, and Fake Identity. The same paper also organized the countermeasures in terms of detection techniques (peer monitoring, information monitoring, policy monitoring, activity monitoring, and attack modeling) and prevention approaches (encryption, access control policies, behavior policies, agent-oriented software engineering, and language-based security). Other related threat taxonomies include the top 25 software vulnerabilities [175] and

dependability threats and failure types [12] that are a superset of security threats and failures.

- In addition to understanding the attacks, it is equally important to understand the objectives we would like to achieve when it comes to software self-protection. A common "CIA" model from the security community defines Confidentiality, Integrity, and Availability as the main security objectives for information systems, as used in [137], [70], and [31].

- Software systems use a variety of techniques to mitigate security threats to achieve the CIA objectives. In addition to those countermeasures catalogued in [19], Sundaram [168] provided a good introduction and categorization on intrusion detection techniques, an important research area related to self-protection. Kumar et al. [95] provided a good survey of Artificial Intelligence (AI) techniques for intrusion detection.

- A number of surveys focused on organizing and classifying security patterns. Konrad et al., for example, uses metrics such as purpose (creational, structural, and behavioral) and abstraction level (network, host, application) [91]. A similar effort by Hafiz et al. [70] proposed other ways to organize security patterns, many of which are applicable to classifying self-protection approaches.

Even though these generic surveys on security attacks, objectives, techniques and patterns are helpful, they do not specifically apply to software self-protection. Four other surveys, however, offer more pertinent insight into how software systems adapt to security threats: First, Elkhodary and Whittle [52] provided a good survey on adaptive security mechanisms. It builds on top of the taxonomy of computational paradigms defined in [150], but adds additional dimensions such as reconfiguration scale and conflict handling. These dimensions are certainly applicable to self-protection systems in general; however, the paper's focus is primarily on the application layer. Secondly, Nguyen and Sood [126] offered

an up-to-date survey on Intrusion Tolerant Systems (ITS), a class of self-protecting systems that focus on continued system operations even in the presence of intrusion attacks. ITS architectures are often based on fault tolerance techniques. Some research efforts identified in this paper are also covered in my analysis in Section 5.4. As correctly pointed out by the authors, these approaches are by no means mutually exclusive and may be used together. Thirdly, Stakhanova et al. [166] and Shameli-Sendi et al. [157] surveyed a different class of systems called Intrusion Response Systems (IRS) that focus on dynamic response mechanisms once an intrusion has been detected. Both surveys proposed an IRS taxonomy that included dimensions such as adjustment ability (adaptive vs. non-adaptive), response selection (static, dynamic, or cost-sensitive), and response type (proactive vs. reactive), which overlap to some extent with my self-protection taxonomy. Cost-sensitive response selection, in particular, corroborated with a similar trend I have identified in my survey.

Even though ITS and IRS have moved beyond traditional static and human-driven security mechanisms, they are still intrusion-centric and perimeter based and as such do not yet constitute true self-protection. In fact, none of the four surveys focused specifically on self-protection research in the AC context. Nor did any of them follow the systematic literature review methodology.

11.2 Related Research

Building Software Engineering Models from System Execution Traces

Data mining techniques are increasingly being applied in the software engineering domain to improve software productivity and quality [197]. The datasets of interest includes execution sequences, call graphs, and text (such as bug reports and software documentation). One body of research, for instance, focuses on mining software specifications — frequent patterns that occur in execution traces [108], which is similar to our problem but the focus is on mining API call usages for purposes such as bug detection, not for self-adaptation; their techniques (such as libSVM) are also different.

The dataset of interest for my dissertation is system execution sequences. Software engineering research has actively focused on mining behavioral models automatically from system execution traces ([42, 94, 109], to name a few). Cook et al. [42] use the event data generated by a software process to discover the formal sequential model of that process. In a subsequent work [43], they have extended their work to use the event traces for a concurrent system to build a concurrency model of it. Gaaloul et al. [63] discover the implicit orchestration protocol behind a set of web services through structural web service mining of the event logs and express them explicitly in terms of BPEL. Motahari-Nezhad et al. [121] present an algorithmic approach for correlating individual events, which are scattered across several systems and data sources, semi-automatically. They use these correlations to find the events that belong to the same business process execution instance. Wen et al. [190] use the start and end of transactions from the event log to build petri-nets corresponding to the processes of the system.

Recent efforts have also expanded to modeling user behavior [67] and concurrent systems [18]. However very few of these efforts focused on security. Only recently did we start to see adaptive security approaches tackling application-level attacks including insider threats, as seen in [13] for instance, which employs dynamically generated access control models [67].

To my knowledge, except our recent work [24, 55, 202], little or no previous work has used mining of execution log to understand the dynamic behavior of the system for the purpose of self-adaptation.

Data Mining in the Security Domain

From the techniques perspective, ARMOUR joins a large body of research in applying data mining methods to the security domain, especially those for anomaly detection (e.g. those in the [33] survey). Much of existing research, however, centered around (a) intrusion detection, especially at network and host levels (e.g., [101] among many others), and (b) malware/virus detection for source code and executables (e.g., [156]). Among those, several

efforts share my approach of unsupervised learning, i.e., using unlabeled or "noisy" training data. Portnoy et al. [140], for example, used a distance-based clustering for detecting network intrusions. Lane and Brodley used unsupervised machine learning based on a similarity measure to classify user behavior in UNIX command shells [98]. Eskin et al. developed a geometric framework that projects unlabeled data to a high-dimensional feature space before applying clustering algorithms to detect anomalies in sparse regions of the feature space [56]. Kwitt and Hofmann used Principle Component Analysis (PCA) based techniques for unsupervised anomaly detection on KDD Cup 1999 data, but its external validity is yet unclear [96].

Still others used mining algorithms such as Support Vector Machines (SVM) [89], Hidden Markov Models (HMM) [189], ensemble based learning [135], graph mining [41], etc. Little research, however, has focused on detecting malicious behavior at the architecture/component level. I believe detecting malicious behavior at the architectural-level is a prerequisite for developing self-protection mechanisms that modify the system's architecture to mitigate the security threats. As mentioned earlier in the thesis, architecture-level anomalies tend to provid more clues to the strategy and intent of the attacks and therefore are more informative and actionable to system administrators.

Anomaly Detection Based on a Normal Behavior Model

The idea of detecting security anomalies based on a model of the system's "normal" behavior is by no means a new one. Early research in the nineties ([61, 78, 98] and many others) exploited the correlation of short sequences of system calls; a sequence that falls outside of normal call patterns may indicate abnormal behavior. Warrender et al. [189], in particular, evaluated 4 methods including enumerating sequences [61, 78], frequent sequences [125], classification rules [102], and HMM, with adequate performances. Their effort, however, indicated no single method consistently gave the best results, and the performance was often dependent on the complexity of the system call traces and their environments.

Later approaches used other constructs such as method call profiles based on dynamic

sandboxing [83], finite state automata based on static program analysis [186], program execution paths extracted from call stack information during normal program runs [59], execution graphs [64], process creation trees [97], or call graphs and calling context trees of cloud-based applications [7].

Our ARMOUR framework adopts the same strategy of buidling normal behavior models, but differs from the above approaches in that: (a) these approaches are host-based, i.e. mining metadata of individual processes or programs, while ARMOUR looks at abstract software component interactions that may span across multiple process spaces or even multiple hosts; (b) many require supervised training whereas ARMOUR can run unsupervised; (c) most assume explicitly or implicitly that normal program behavior is deterministic and stable, where ARMOUR assumes the behavior for an interactive system is inherently fluid and user-driven, and hence continually updates the model based on recent system execution traces.

The normal behavior-based anomaly detection methods, especially the unsupervised ones, often suffers from the problem of false positives, many of which may be uninteresting yet cause unnecessary human attention, thus diverting precious resources away from true threats. Song et al. [160] presented an interesting approach called *conditional anomaly detection* that takes advantage of user-provided environmental properties to help find relevant anomalies. Even though their algorithms were statistical and parametric and their evaluation was limited to the KDD network intrusion data, their methodology is similar in spirit to the conditional frequent pattern mining heuristic used in ARMOUR (described in Section 8.2.

Associations Mining Based Approaches

A number of past research efforts share my choice of the mining technique, that is, associations mining and its derivative, frequent episode mining. Even though the technique is simple compared with more powerful alternatives such as ANN or HMM, it is computationally efficient and fast, making it ideally suited for near real-time anomaly detction.

He et al. [76] also hypothesized anomalous data record occurs in fewer frequent itemsets compared to normal data record, and proposed a quantitative measure called Frequent Pattern Outlier Factor (FPOF). Their evaluation, however, was not on security.

Otey et al. [131], for example, used association mining for network intrusion detection at the network interface card (NIC) level. Li et al. [106] uses Apriori-based association mining to build normal usage profiles for local area networks (LAN), assuming usage patterns and user habits are stable. Chan and Mahoney [32] took a slightly different approach with their Learning Rules for Anomaly Detection (LERAD) algorithm; their goal was to generate a set of anomaly rules for anomaly detection as opposed to using the normal model, which is sometimes referred to as netagive pattern mining. Their dataset of interest was also network traffic.

To describe temporal normal behavior, a few researches also turned to frequent episode mining [103, 144]. Their evaluations were also on network intrusion detection, and their implementations were based on the algorithm proposed by Mannila and Toivonen [115] instead of GSP.

The difference that sets ARMOUR apart is that these previous efforts rely on domain-specific features (specifically, TCP/IP protocol attributes such as source and destination IP addresses, ports, flags, etc.), whereas ARMOUR is domain-independent – it can be applied as long as abstraction components are identified and component interactions can be observed.

Chapter 12: Conclusion

In this chapter I summarize the contributions of my research and discuss a number of threats to its validity, as well as some remaining research challenges for future work.

12.1 Contributions

In this thesis I have made the case for the importance of monitoring and assessing the overall security posture of a software system at the *architecture* level in order to detect and mitigate more sophisticated threats that may otherwise go unnoticed using traditional intrusion detection techniques. In Chapter 6, I validated my first hypothesis that it is possible to mine a probabilistic architectural model that captures a software system's normal behavior from its execution history. In Chapter 7 and 8, I introduced the ARMOUR framework that validated my second hypothesis that using an automatically mined architectural model, it is possible to detect a large-class of security threats that violate the normal usage of the software. In Chapter 9, I completed the ARMOUR framework with a pattern-based defensive mechanism that proved software architecture can indeed provide an additional line of defense for effectively mitigating security threats at runtime.

My research makes a number of contributions.

First, I conducted a systematic survey of start-of-the-art self-protection research. To the best of our knowledge, the survey presented in Chapter 5 is the most comprehensive and elaborate investigation of the literature in this topic area to date. Along the way, I also developed a comprehensive taxonomy to characterize self-protection research techniques and methods.

Second, I used data mining of a system's execution history at runtime to understand the dynamic behavior of the system for the purpose of self-adaptation. In particular, the ARM

and GSP mining techniques are used to capture a form of probabilistic component interaction model. This sets this research apart from most of the "models@runtime" techniques that rely on static, manually developed models.

Third, I developed a novel detection algorithm based on the mining model that can effectively detect anomalous and potentially malicious behavior that deviates from the system's normal user behavior. The algorithm is use case-driven and adaptive to the system behavior shifts and load fluctuations. In contrast, most security-related data mining efforts look at low-level intrusions such as system calls and network packets as opposed to "macro-level' system traits.

Finally, I identified and catalogued a set of repeatable self-protection architectural patterns, both structural and behavorial, that have been the most comprehensive to date. These patterns serve as initial building blocks to enhance the overall security of a self-adaptive system in ways that traditional, perimeter-based defense mechanisms couldn't.

From a practitioner's prespective, my research complements existing security mechanisms and brings the following benefits:

- Detection of architectural-level anomalous component interactions can serve as important clues that help system administrators better determine the *intent* of malicious attacks, regardless of how lower-level attack tactics may camouflage or morph themselves;
- Effective against new exploits by undiscovered,"zero-day" threats;
- Practical implementation with unsupervised learning, inline detection, and ability to self-adapt to changes in system load and user behavior drifts at runtime;
- Effective in detecting application-level insider attacks that may be otherwise undetectable by perimeter-focused intrusion detection methods;
- No domain-specific assumptions on the target system, making it suitable for a broad range of applications.

12.2 Threats to Validity

A few possible threats to the validity of our approach deserve additional discussion.

First, the underlying assumption in the current version of our framework is that a single data mining algorithm can process all the events/transactions in the system and build the stochastic component interaction models. This may not be possible, especially when we consider distributed software systems that permeate boundaries of several enterprises. An enterprise may be unwilling to share its internal structure and event logs with an entity that is out of its control for various reasons (e.g., protecting competitive edge, security concerns, etc.). Therefore, a distributed version of our approach may work better, which achieves the same goal by running multiple local data mining algorithms.

The second threat has to do with the space complexity of the mining framework. For a large-scale system with potentially hundreds of components and even larger number of event types, the interactions in the system may be quite diverse. The need to keep minimum support level low for the mining algorithms may cause the number of rules or patterns captured in the model to grow exponentially. For the ARM method, I have mitigated this challenge by using efficient encoding and indexing of the rules (1 million rules, for example, only takes about 50MB of memory) and use effective rule pruning techniques as shown in Section 7.6.4. I have also used the GSP mining method that drastically reduced the number of sequential patterns. For a highly complex system, this challenge can be mitigated further by dividing the system into smaller, more manageable subsystems and applying the ARMOUR framework in a hierarchical fashion.

Third, the ability to identify external use cases or UIEs might be a challenge for certain types of systems. An ad-hoc system with free-form interactions across components (such as a wireless network of sensors), for example, may not be a good candidate for my approach. The ideal candidates for applying the ARMOUR framework would be online enterprise systems that have clear business objectives and process patterns, regardless of application domain.

Last but not the least, all security mechanisms, including ARMOUR, are subject to exploitation once their approaches are known to attackers. In ARMOUR's case, for instance, an attacker could slowly increase the frequency of the anomaly events, to a point that they start to "blend in" with normal usage, as mentioned in earlier sensitivity analysis. To mitigate this threat in practice, the ARMOUR framework should be used in conjunction with conventional security mechanisms such as DoS prevention, application firewalls, blacklists / whitelists, etc., so that ARMOUR can focus on application-level, hard-to-detect threats.

12.3 Future Work

My future work will attempt to make the framework more robust, such as improving computational efficiency using cloud computing techniques and integrating ARMOUR with pattern-based self-protection methods (such as those proposed in Chapter 9) to auto-respond to threats at runtime.

The arena of software system self-protection is a fertile ground for continued research. The following are just a few suggested areas:

- *Quantifying security.* One of the difficulties in automatically making adaptation decisions is the lack of established and commonly accepted metrics for the quantification of security. Good security metrics are needed to enable comparison of candidate adaptations, evaluate the effect adaptations have on security properties, and quantify overall system security posture. However, there are few security metrics that can be applied at an architectural level. Architectural-level metrics are preferred because they reflect the system security properties affected by adaptations. Such metrics could include measures to classify security based on applied adaptations and evaluate the impact of adaptations on certain security properties (e.g., attack surface, number least privilege violations).

- *Quality attribute tradeoffs.* In fielded systems security must be considered with other,

possibly conflicting, quality attributes. Rainbow makes tradeoffs between quality attributes with each adaptation, selecting an adaptation that maximizes the overall utility of the system. Principled mechanisms are needed to evaluate the impacts of these tradeoffs as the system changes. Consider that almost all self-protection patterns described in Chapter 9 come at the expense of other quality attributes (e.g. response time, availability). Mechanisms to automatically evaluate competing quality attributes is critical for effective self adaptation. Software architecture is the appropriate medium for evaluating such tradeoffs automatically because it provides a holistic view of the system. Rainbow, for example, reasons about a multi-dimensional utility function, where each dimension represents the user's preferences with respect to a particular quality attribute, to select an appropriate strategy.

- *Formalizing the Self-Protection Patterns.* Continued research is needed for analyzing and systematically cataloguing ABSP patterns and making them available to the community. To make the ABSP patterns more consistent and machine-readable, they also need to be formally specified in architecture definition languages (such as UML or ACME).

- *Protecting the self-protection logic.* Most of the research to date has assumed the self-protection logic itself is immune from security attacks. One of the reasons for this simplifying assumption is that prior techniques have not achieved a disciplined split between the protected subsystem and the protecting subsystem. In my approach, the inversion of dependency and clear separation of application logic from the architectural layer present an opportunity to address this problem. One opportunity, for example, is to leverage techniques such as virtualization, thereby reducing the likelihood of the meta-level subsystem being compromised by the same threats targeted at the application logic.

Appendix A: Survey Result Details

The evaluation matrix below contains the detailed survey results for Chapter 5.

Source	Self-Prot. Levels			Depths of Defense					Life cycle Focu		Security Goals			Meta-Level Sep.			Theoretical Foundation					Meta-Level DM			Control Topology			Response Timing			Enforce. Locale			Self-Protection Patterns										Validation Method				App.		Rep.	
	Monitor & Detect	Respond & Protect	Plan & Prevent	Network	Host	Middleware	Application	Depth-Ind.	Dev Time	Runtime	Confidentiality	Integrity	Availability	No Separation	Partial Sep.	Complete Sep.	Formal Models	Heuristic Rules	Optimization	Prob. Models	Machine Learning	Single Strategy	Multi-Strategy	Cost-sensitive	Local Only	Globally Centr.	Globally Decentr.	Reactive	Proactive	Timing-Neutral	System Boundary	System Internals	Locale-neutral	Protective Wrapper	Agmt-based Redun	Impl. Diversity	S/W Rejuvenation	Protective Recomp	Attack Containmen	Reconfig on Reflex	Ctr-measure Broker	Aspect-Orientation	Artificial Immun.	Empirical	Proof	Simulation	None	High	Low	High	Low
[Abie et al. 2008]	x	x	x	x	x	x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Abie 2009]	x	x	x	x	x	x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Savola and Heinonen 2010]	x	x	x	x	x	x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Adnane et al. 2008]	x			x	x	x			x	x	x	x	x			x	x	x				x				x			x			x											x				x		x	
[Alampalayam and Kumar 2003]	x	x	x	x	x	x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Alia and Lacoste 2008]	x	x				x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Alia et al. 2010]	x	x				x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Al-Nashif et al. 2008]	x	x		x					x	x	x	x	x			x		x	x			x				x			x			x											x				x		x	
[Atighetchi et al. 2003]	x	x		x	x	x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Atighetchi et al. 2004]	x	x		x	x	x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Balepin et al. 2003]	x	x			x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Ben Mahmoud et al. 2010]	x	x		x					x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Benjamin et al. 2008]	x	x		x					x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Blount et al. 2011]	x				x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Burns et al. 2001]	x	x			x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Casola et al. 2008]	x	x		x					x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Castro and Liskov 2002]	x	x			x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Chigan et al. 2005]	x	x		x					x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Chong et al. 2005]	x	x		x	x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Pal et al. 2007]	x	x		x	x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Costa et al. 2008]	x	x			x	x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Cox et al. 2007]	x			x					x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Crosbie and Spafford 1995]	x				x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[de Oliveira et al. 2011]	x	x		x					x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[De Palma et al. 2012]	x			x	x	x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Debar et al. 2007]	x	x	x	x	x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Dragoni et al. 2009]	x	x				x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[English et al. 2006]	x				x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Erlingsson and Schneider 2000]	x	x		x					x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Fayssal et al. 2008]	x	x				x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Feiertag et al. 2000]	x	x		x	x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Foo et al. 2005]	x	x				x			x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Wu et al. 2007]	x	x		x	x				x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Garlan et al. 2004]	x	x	x						x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Cheng et al. 2006]	x	x	x					x	x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Gelenbe and Loucas 2007]	x	x		x					x	x	x	x	x			x		x				x				x			x			x											x				x		x	
[Ghosh et al. 1998]	x								x		x	x						x				x				x			x			x											x				x		x	
[Ghosh and Voas 1999]	x								x		x	x						x				x				x			x			x											x				x		x	

Source	Self-Prot. Levels			Depths of Defense					Life cycle Focu	Security Goals			Meta-Level Sep.			Theoretical Foundation					Meta-Level DM			Control Topology			Response Timing			Enforce. Locale			Self-Protection Patterns									Validation Method				App.		Rep.	
	Monitor & Detect	Respond & Protect	Plan & Prevent	Network	Host	Middleware	Application	Depth-Ind.	Dev Time	Runtime	Confidentiality	Integrity	Availability	No Separation	Partial Sep.	Complete Sep.	Formal Models	Heuristic Rules	Optimization	Prob. Models	Machine Learning	Single Strategy	Multi-Strategy	Cost-sensitive	Local Only	Globally Centr.	Globally Decentr.	Reactive	Proactive	Timing-Neutral	System Boundary	System Internals	Locale-neutral	Protective Wrapper	Agmt-based Redun	Impl. Diversity	S/W Rejuvenation	Protective Recomp	Attack Containmen	Reconfig on Reflex	Ctr-measure Broker	Aspect-Orientation	Artificial Immun.	Empirical	Proof	Simulation	None	High	Low	High	Low
[Goel et al. 2005]	x	x			x					x		x				x		x				x			x			x			x											x				x		x	
[Jain et al. 2008]	x	x			x					x		x				x		x				x			x			x			x											x				x		x	
[Guttman and Herzog 2005]	x	x	x	x						x	x	x			x		x					x						x				x											x				x		x	
[Hashii et al. 2000]	x				x		x		x	x	x	x						x				x			x			x			x											x				x		x	
[He et al. 2010a]	x	x	x	x	x					x	x	x	x					x				x			x			x			x											x				x		x	
[He et al. 2010b]	x	x	x	x	x					x	x	x	x					x				x			x			x			x											x				x		x	
[Hinton et al. 1999]	x	x	x				x			x					x									x				x																			x			
[Huang et al. 2006]	x	x			x					x		x				x		x				x			x				x			x															x			
[Nagarajan et al. 2011]	x	x			x					x		x				x		x				x			x				x			x															x			
[Jabbour and Menasce 2008]	x	x					x			x	x		x					x				x			x				x			x															x			
[Jabbour and Menasce 2009]	x	x					x			x	x		x					x				x			x				x			x															x			
[Jansen et al. 2008]	x	x			x					x	x	x	x					x				x			x				x			x															x			
[Jean et al. 2007]	x					x				x		x		x				x			x	x					x		x			x															x			
[Kephart et al. 1997]	x	x	x		x		x		x	x	x	x				x		x				x			x				x			x															x			
[White et al. 1999]	x	x	x		x		x		x	x	x	x				x		x				x			x				x			x															x			
[Labraoui et al. 2010]	x								x	x			x			x		x				x			x		x		x			x															x			
[Lamprecht and van Moorsel	x	x		x						x		x			x			x				x			x				x			x															x			
[Lamprecht and van Moorsel	x	x		x						x	x	x			x			x				x			x				x			x															x			
[Laureano et al. 2007]	x	x			x					x			x			x		x			x	x			x				x			x															x			
[Lee et al. 2002]	x			x	x					x	x	x			x			x			x	x			x				x			x															x			
[Liang and Sekar 2005]	x						x		x	x	x	x						x				x			x				x			x															x			
[Lim et al. 2009]	x	x					x		x	x	x	x			x			x	x		x		x		x				x			x															x			
[Lorenzoli et al. 2007]	x	x					x		x	x		x		x				x	x			x			x				x			x															x			
[Malek et al. 2009]	x	x	x					x	x	x	x	x	x					x	x			x				x			x			x															x			
[Menasce et al. 2011]	x	x	x					x	x	x	x	x				x		x	x			x				x			x			x															x			
[Maximilien and Singh 2004]	x	x					x			x			x		x			x				x			x				x			x															x			
[Montangero and Semini 2004]	x	x		x						x	x	x				x	x	x				x			x				x			x															x			
[Mouelhi et al. 2008]		x	x					x	x	x	x	x			x			x				x			x				x			x															x			
[Morin et al. 2010]		x	x					x	x	x	x	x				x		x				x			x				x			x															x			
[Musman and Flesher 2000]	x	x		x	x					x	x	x			x			x				x				x			x			x															x			
[Okhravi et al. 2010]	x	x		x	x		x		x	x		x				x		x				x			x				x			x															x			
[Okhravi et al. 2012]	x	x		x	x		x		x	x		x				x		x				x			x				x			x															x			
[Ostrovsky and Yung 1991]		x				x				x		x				x	x	x				x			x				x			x															x			
[Porras and Neumann 1997]	x	x		x						x	x	x				x		x				x			x				x			x															x			
[Neumann and Porras 1999]	x	x		x						x	x	x				x		x				x			x				x			x															x			
[Portokalidis and Bos 2006]	x	x		x						x		x				x		x				x			x				x			x															x			
[Raissi 2006]	x	x			x					x	x					x		x	x			x			x				x			x															x			
[Rawat and Saxena 2009]	x	x		x						x		x				x		x				x				x			x			x															x			

Source	Self-Prot. Levels			Depths of Defense				Life cycle Focu	Security Goals			Meta-Level Sep.			Theoretical Foundation					Meta-Level DM			Control Topology			Response Timing			Enforce. Locale			Self-Protection Patterns									Validation Method				App.		Rep.	
	Monitor & Detect	Respond & Protect	Plan & Prevent	Network	Host	Middleware	Application	Depth-Ind.	Confidentiality	Integrity	Availability	No Separation	Partial Sep.	Complete Sep.	Formal Models	Heuristic Rules	Optimization	Prob. Models	Machine Learning	Single Strategy	Multi-Strategy	Cost-sensitive	Local Only	Globally Centr.	Globally Decentr.	Reactive	Proactive	Timing-Neutral	System Boundary	System Internals	Locale-neutral	Protective Wrapper	Agmt-based Redun	Impl. Diversity	S/W Rejuvenation	Protective Recomp	Attack Containmen	Reconfig on Reflex	Ctr-measure Broker	Aspect-Orientation	Artificial Immun.	Empirical	Proof	Simulation	None	High	Low	High	Low
[Reiser and Kapitza 2007a]		x	x		x						x			x		x				x				x			x			x										x				x		x	
[Reiser and Kapitza 2007b]			x		x						x			x		x				x				x			x			x										x				x		x	
[Reynolds et al. 2002]		x	x		x					x	x			x		x				x				x			x			x										x				x		x	
[Reynolds et al. 2003]		x	x		x					x	x			x		x				x				x			x			x										x				x		x	
[Saxena et al. 2007]		x	x	x				x						x		x								x			x			x										x				x		x	
[He and Lacoste 2008]		x	x	x				x		x	x			x		x					x			x			x			x										x				x		x	
[Sibai and Menasce 2011]		x	x		x					x	x			x		x				x				x			x			x										x				x		x	
[Sibai and Menasce 2012]		x	x		x					x	x			x		x				x				x			x			x										x				x		x	
[Sousa et al. 2006]		x	x		x	x				x	x			x		x		x			x			x			x			x										x				x		x	
[Sousa et al. 2007]		x	x		x	x				x	x			x		x		x			x			x			x			x										x				x		x	
[Sousa et al. 2010]		x	x		x	x				x	x			x		x		x			x			x			x			x										x				x		x	
[Sakhanova 2007]		x	x		x					x	x		x			x	x			x		x					x			x										x				x		x	
[Strasburg 2009]		x	x		x					x	x		x			x	x			x		x					x			x										x				x		x	
[Swimmer 2006]		x	x	x						x			x			x				x				x			x			x										x				x		x	
[Taddeo and Ferrante 2009]			x	x						x	x	x				x	x			x				x			x			x										x				x		x	
[Tan et al. 2008]		x					x			x	x				x	x				x				x			x			x										x				x		x	
[Tang and Yu 2008]		x	x		x					x	x					x				x				x			x			x										x				x		x	
[Uribe and Cheung 2004]			x	x						x	x				x	x				x				x			x			x										x				x		x	
[Valdes et al. 2004]		x	x			x				x	x		x			x				x		x		x			x			x										x				x		x	
[Verissimo et al. 2006]		x	x		x	x				x	x			x		x				x				x			x			x										x				x		x	
[Vikram et al. 2009]		x					x			x				x		x				x				x			x			x										x				x		x	
[Wang et al. 2003]		x	x	x	x					x	x			x		x				x				x			x			x										x				x		x	
[Wang et al. 2009]		x	x				x			x			x			x				x				x			x			x										x				x		x	
[Xiao et al. 2007]		x	x			x				x	x			x		x				x				x			x			x										x				x		x	
[Xiao 2008]		x	x			x				x	x			x		x				x				x			x			x										x				x		x	
[Yau et al. 2006]		x	x				x			x	x			x		x				x				x			x			x										x				x		x	
[Yu et al. 2007]		x			x					x	x			x		x				x				x			x			x										x				x		x	
[Yu et al. 2008]		x								x	x			x		x				x				x			x			x										x				x		x	
[Zhang and Shen 2009]		x	x	x	x	x				x	x			x		x				x				x			x			x										x				x		x	
[Zhang et al. 2011]		x	x	x	x	x				x	x			x		x				x				x			x			x										x				x		x	
[Zou et al. 2002]		x	x		x					x	x				x	x				x				x			x			x										x				x		x	

Bibliography

Bibliography

[1] Abie, H. Adaptive security and trust management for autonomic message-oriented middleware. In *Mobile Adhoc and Sensor Systems, 2009. MASS '09. IEEE 6th International Conference on* (2009), pp. 810–817.

[2] Abie, H., Dattani, I., Novkovic, M., Bigham, J., Topham, S., and Savola, R. GEMOM - Significant and Measurable Progress beyond the State of the Art. In *3rd International Conference on Systems and Networks Communications, 2008. ICSNC '08* (Oct. 2008), pp. 191 –196.

[3] Adnane, A., de Sousa, Jr., R. T., Bidan, C., and M, L. Autonomic trust reasoning enables misbehavior detection in OLSR. In *Proceedings of the 2008 ACM symposium on Applied computing* (New York, NY, USA, 2008), SAC '08, ACM, pp. 2006–2013.

[4] Agrawal, R., and Srikant, R. Fast Algorithms for Mining Association Rules in Large Databases. In *20th International Conference on Very Large Data Bases* (1994), Morgan Kaufmann, Los Altos, CA, pp. 478–499.

[5] Al-Nashif, Y., Kumar, A., Hariri, S., Qu, G., Luo, Y., and Szidarovsky, F. Multi-Level Intrusion Detection System (ML-IDS). In *International Conference on Autonomic Computing, 2008. ICAC '08* (June 2008), pp. 131 –140.

[6] Alampalayam, S., and Kumar, A. Security model for routing attacks in mobile ad hoc networks. In *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th* (2003), vol. 3, pp. 2122– 2126 Vol.3.

[7] Alsouri, S., Sinschek, J., Sewe, A., Bodden, E., Mezini, M., and Katzenbeisser, S. Dynamic Anomaly Detection for More Trustworthy Outsourced Computation. In *Information Security*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, D. Gollmann, and F. C. Freiling, Eds., vol. 7483. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 168–187.

[8] Amazon Web Services. Amazon web services(AWS) cloudformation, Jan. 2015.

[9] Atighetchi, M., and Pal, P. From Auto-adaptive to Survivable and Self-Regenerative Systems Successes, Challenges, and Future. In *Eighth IEEE International Symposium on Network Computing and Applications, 2009. NCA 2009* (July 2009), pp. 98 –101.

[10] Atighetchi, M., Pal, P., Jones, C., Rubel, P., Schantz, R., Loyall, J., and Zinky, J. Building auto-adaptive distributed applications: the QuO-APOD experience. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings* (May 2003), pp. 104 – 109.

[11] Atighetchi, M., Pal, P., Webber, F., Schantz, R., Jones, C., and Loyall, J. Adaptive cyberdefense for survival and intrusion tolerance. *IEEE Internet Computing 8*, 6 (Dec. 2004), 25 – 33.

[12] Aviienis, A., Laprie, J.-C., and Randell, B. Dependability and Its Threats: A Taxonomy. In *Building the Information Society*, R. Jacquart, Ed., vol. 156 of *IFIP International Federation for Information Processing*. Springer Boston, 2004, pp. 91–120.

[13] Bailey, C., Montrieux, L., de Lemos, R., et al. Run-time Generation, Transformation, and Verification of Access Control Models for Self-protection. In *SEAMS 2014*, ACM, pp. 135–144.

[14] Balepin, I., Maltsev, S., Rowe, J., and Levitt, K. Using Specification-Based Intrusion Detection for Automated Response. In *Recent Advances in Intrusion Detection*, G. Vigna, C. Kruegel, and E. Jonsson, Eds., vol. 2820 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003, pp. 136–154.

[15] Barna, C., Shtern, M., Smit, M., Tzerpos, V., and Litoiu, M. Model-based adaptive DoS attack mitigation. In *2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2012), pp. 119–128.

[16] Ben Mahmoud, M., Larrieu, N., Pirovano, A., and Varet, A. An adaptive security architecture for future aircraft communications. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th* (2010), pp. 3.E.2–1–3.E.2–16.

[17] Bencsth, B., Pk, G., Buttyn, L., and Flegyhzi, M. Duqu: Analysis, Detection, and Lessons Learned. In *ACM European Workshop on System Security (EuroSec)* (2012), vol. 2012.

[18] Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 468–479.

[19] Bijani, S., and Robertson, D. A review of attacks and security approaches in open multi-agent systems. *Artificial Intelligence Review* (May 2012), 1–30.

[20] Blount, J., Tauritz, D., and Mulder, S. Adaptive Rule-Based Malware Detection Employing Learning Classifier Systems: A Proof of Concept. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual* (July 2011), pp. 110 –115.

[21] Bobroff, N., Kochut, A., and Beaty, K. Dynamic placement of virtual machines for managing SLA violations. In *IFIP/IEEE Int'l Symp. on Integrated Network Management* (Munich, Germany, May 2007), pp. 119–128.

[22] Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software 80*, 4 (Apr. 2007), 571–583.

[23] Burns, J., Cheng, A., Gurung, P., Rajagopalan, S., Rao, P., Rosenbluth, D., Surendran, A., and Martin, D.M., J. Automatic management of network security policy. In *DARPA Information Survivability Conference Exposition II, 2001. DISCEX '01. Proceedings* (2001), vol. 2, pp. 12 –26 vol.2.

[24] Canavera, K. R., Esfahani, N., and Malek, S. Mining the Execution History of a Software System to Infer the Best Time for its Adaptation. In *20th International Symposium on the Foundations of Software Engineering* (Nov. 2012).

[25] Cappelli, D. M., Moore, A. P., and Trzeciak, R. F. *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes*, 1st ed. Addison-Wesley Professional, 2012.

[26] Casalicchio, E., Menasc, D. A., and Aldhalaan, A. Autonomic resource provisioning in cloud systems with availability goals. In *ACM Cloud and Autonomic Computing Conf.* (Miami, Florida, Aug. 2013), pp. 1:1–1:10.

[27] Casanova, P., Garlan, D., Schmerl, B., and Abreu, R. Diagnosing architectural run-time failures. To appear in SEAMS, 2013.

[28] Casanova, P., Schmerl, B., Garlan, D., and Abreu, R. Architecture-based Run-time Fault Diagnosis. In *Proceedings of the 5th European Conference on Software Architecture* (2011).

[29] Casola, V., Mancini, E. P., Mazzocca, N., Rak, M., and Villano, U. Self-optimization of secure web services. *Computer Communications 31*, 18 (Dec. 2008), 4312–4323.

[30] Castro, M., and Liskov, B. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov. 2002), 398–461.

[31] Cavalcante, R. C., Bittencourt, I. I., da Silva, A. P., Silva, M., Costa, E., and Santos, R. A survey of security in multi-agent systems. *Expert Systems with Applications 39*, 5 (Apr. 2012), 4835–4846.

[32] Chan, P. K., Mahoney, M. V., and Arshad, M. H. Learning Rules and Clusters for Anomaly Detection in Network Traffic. In *Managing Cyber Threats*, V. Kumar, J. Srivastava, and A. Lazarevic, Eds., no. 5 in Massive Computing. Springer US, 2005, pp. 81–99. DOI: 10.1007/0-387-24230-9_3.

[33] Chandola, V., Banerjee, A., and Kumar, V. Anomaly Detection: A Survey. *ACM Comput. Surv. 41*, 3 (July 2009), 15:1–15:58.

[34] Cheng, B. H. C., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi,

V., Karsai, G., Kienle, H. M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Mller, H. A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds., vol. 5525. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 1–26.

[35] Cheng, S.-W., and Garlan, D. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software 85*, 12 (Dec. 2012), 2860–2875.

[36] Cheng, S.-W., Garlan, D., and Schmerl, B. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems* (New York, NY, USA, 2006), SEAMS '06, ACM, pp. 2–8.

[37] Cheng, S.-W., Garlan, D., and Schmerl, B. Evaluating the effectiveness of the Rainbow self-adaptive system. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09* (May 2009), pp. 132 –141.

[38] Chess, D. M., Palmer, C. C., and White, S. R. Security in an autonomic computing environment. *IBM Systems Journal 42*, 1 (2003), 107 –118.

[39] Chigan, C., Li, L., and Ye, Y. Resource-aware self-adaptive security provisioning in mobile ad hoc networks. In *Wireless Communications and Networking Conference, 2005 IEEE* (2005), vol. 4, pp. 2118– 2124 Vol. 4.

[40] Chong, J., Pal, P., Atigetchi, M., Rubel, P., and Webber, F. Survivability architecture of a mission critical system: the DPASA example. In *Computer Security Applications Conference, 21st Annual* (Dec. 2005), pp. 10 pp. –504.

[41] Christodorescu, M., Jha, S., and Kruegel, C. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference* (2008), ACM, pp. 5–14.

[42] Cook, J. E., and Wolf, A. L. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM) 7*, 3 (1998), 215–249.

[43] Cook, J. E., and Wolf, A. L. Event-based Detection of Concurrency. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 1998), SIGSOFT '98/FSE-6, ACM, pp. 35–45.

[44] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. Vigilante: End-to-end containment of Internet worm epidemics. *ACM Trans. Comput. Syst. 26*, 4 (Dec. 2008), 9:1–9:68.

[45] Councill, B., and Heineman, G. T. Definition of a Software Component and Its Elements. In *Component-based Software Engineering*, G. T. Heineman and W. T.

Councill, Eds. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001, pp. 5–19.

[46] Crosbie, M., and Spafford, G. Active Defense of a Computer System Using Autonomous Agents. Tech. rep., 1995.

[47] De Palma, N., Hagimont, D., Boyer, F., and Broto, L. Self-Protection in a Clustered Distributed System. *Parallel and Distributed Systems, IEEE Transactions on 23*, 2 (2012), 330–336.

[48] Debar, H., Thomas, Y., Cuppens, F., and Cuppens-Boulahia, N. Enabling automated threat response through the use of a dynamic security policy. *Journal in Computer Virology 3*, 3 (2007), 195–210.

[49] Dittmann, J., Karpuschewski, B., Fruth, J., Petzel, M., and M"under, R. An exemplary attack scenario: threats to production engineering inspired by the Conficker worm. In *Proceedings of the First International Workshop on Digital Engineering* (New York, NY, USA, 2010), IWDE '10, ACM, pp. 25–32.

[50] Dragoni, N., Massacci, F., and Saidane, A. A self-protecting and self-healing framework for negotiating services and trust in autonomic communication systems. *Computer Networks 53*, 10 (July 2009), 1628–1648.

[51] Elia, I., Fonseca, J., and Vieira, M. Comparing SQL Injection Detection Tools Using Attack Injection: An Experimental Study. In *2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)* (Nov. 2010), pp. 289–298.

[52] Elkhodary, A., and Whittle, J. A Survey of Approaches to Adaptive Application Security. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2007. ICSE Workshops SEAMS '07* (May 2007), p. 16.

[53] English, C., Terzis, S., and Nixon, P. Towards self-protecting ubiquitous systems: monitoring trust-based interactions. *Personal and Ubiquitous Computing 10*, 1 (2006), 50–54.

[54] Erlingsson, U., and Schneider, F. SASI enforcement of security policies: a retrospective. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings* (2000), vol. 2, pp. 287 –295 vol.2.

[55] Esfahani, N., Yuan, E., Canavera, K. R., and Malek, S. Inferring software component interaction dependencies for adaptation support. vol. 10. ACM Trans. on Autonomous and Adaptive Systems, 2016.

[56] Eskin, E., Arnold, A., Prerau, M., Portnoy, L., and Stolfo, S. A Geometric Framework for Unsupervised Anomaly Detection. In *Applications of Data Mining in Computer Security*, D. Barbar17 and S. Jajodia, Eds., no. 6 in Advances in Information Security. Springer US, Jan. 2002, pp. 77–101.

[57] Fayssal, S., Alnashif, Y., Kim, B., and Hariri, S. A proactive wireless self-protection system. In *Proceedings of the 5th international conference on Pervasive services* (New York, NY, USA, 2008), ICPS '08, ACM, pp. 11–20.

[58] Feiertag, R., Rho, S., Benzinger, L., Wu, S., Redmond, T., Zhang, C., Levitt, K., Peticolas, D., Heckman, M., Staniford, S., and McAlerney, J. Intrusion detection inter-component adaptive negotiation. *Computer Networks 34*, 4 (Oct. 2000), 605–621.

[59] Feng, H., Kolesnikov, O., Fogla, P., Lee, W., and Gong, W. Anomaly detection using call stack information. In *2003 Symposium on Security and Privacy, 2003. Proceedings* (May 2003), pp. 62–75.

[60] Foo, B., Wu, Y.-S., Mao, Y.-C., Bagchi, S., and Spafford, E. ADEPTS: adaptive intrusion response using attack graphs in an e-commerce environment. In *International Conference on Dependable Systems and Networks, 2005. DSN 2005. Proceedings* (July 2005), pp. 508 – 517.

[61] Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T. A sense of self for Unix processes. In *, 1996 IEEE Symposium on Security and Privacy, 1996. Proceedings* (May 1996), pp. 120–128.

[62] Frincke, D., Wespi, A., and Zamboni, D. From intrusion detection to self-protection. *Computer Networks 51*, 5 (Apr. 2007), 1233–1238.

[63] Gaaloul, W., Bana, K., and Godart, C. Log-based mining techniques applied to Web service composition reengineering. *Service Oriented Computing and Applications 2*, 2-3 (May 2008), 93–110.

[64] Gao, D., Reiter, M. K., and Song, D. Gray-box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 318–329.

[65] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer 37*, 10 (Oct. 2004), 46 – 54.

[66] Garlan, D., Monroe, R. T., and Wile, D. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, G. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[67] Ghezzi, C., Pezz, M., Sama, M., and Tamburrelli, G. Mining behavior models from user-intensive web applications. In *ICSE* (2014), pp. 277–287.

[68] Ghosh, A., O'Connor, T., and McGraw, G. An automated approach for identifying potential vulnerabilities in software. In *1998 IEEE Symposium on Security and Privacy, 1998. Proceedings* (May 1998), pp. 104 –114.

[69] Ghosh, A. K., and Voas, J. M. Inoculating software for survivability. *Commun. ACM 42*, 7 (July 1999), 38–44.

[70] Hafiz, M., Adamczyk, P., and Johnson, R. Organizing Security Patterns. *IEEE Software 24*, 4 (Aug. 2007), 52 –60.

[71] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. The WEKA data mining software: an update. *SIGKDD Explor. Newsl. 11*, 1 (Nov. 2009), 10–18.

[72] Han, J., Pei, J., and Yin, Y. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record* (2000), vol. 29, ACM, pp. 1–12.

[73] Hashii, B., Malabarba, S., Pandey, R., and Bishop, M. Supporting reconfigurable security policies for mobile programs. *Computer Networks 33*, 16 (June 2000), 77–93.

[74] He, R., and Lacoste, M. Applying component-based design to self-protection of ubiquitous systems. In *Proceedings of the 3rd ACM workshop on Software engineering for pervasive services* (2008), pp. 9–14.

[75] He, R., Lacoste, M., and Leneutre, J. A Policy Management Framework for Self-Protection of Pervasive Systems. In *2010 Sixth International Conference on Autonomic and Autonomous Systems (ICAS)* (Mar. 2010), pp. 104 –109.

[76] He, Z., Xu, X., Huang, J. Z., and Deng, S. A Frequent Pattern Discovery Method for Outlier Detection. In *Advances in Web-Age Information Management*, Q. Li, G. Wang, and L. Feng, Eds., no. 3129 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, July 2004, pp. 726–732. DOI: 10.1007/978-3-540-27772-9_80.

[77] Hinton, H., Cowan, C., Delcambre, L., and Bowers, S. SAM: Security Adaptation Manager. In *Computer Security Applications Conference, 1999. (ACSAC '99) Proceedings. 15th Annual* (1999), pp. 361–370.

[78] Hofmeyr, S. A., Forrest, S., and Somayaji, A. Intrusion Detection Using Sequences of System Calls. *J. Comput. Secur. 6*, 3 (Aug. 1998), 151–180.

[79] Huang, Y., Arsenault, D., and Sood, A. Closing cluster attack windows through server redundancy and rotations. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06* (May 2006), vol. 2, pp. 12 pp. –21.

[80] Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. Software rejuvenation: analysis, module and applications. In *, Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers* (June 1995), pp. 381 –390.

[81] Huebscher, M. C., and McCann, J. A. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv. 40*, 3 (Aug. 2008), 7:1–7:28.

[82] Igure, V., and Williams, R. Taxonomies of attacks and vulnerabilities in computer systems. *IEEE Communications Surveys Tutorials 10*, 1 (2008), 6 –19.

[83] Inoue, H., and Forrest, S. Anomaly Intrusion Detection in Dynamic Execution Environments. In *Proceedings of the 2002 Workshop on New Security Paradigms* (New York, NY, USA, 2002), NSPW '02, ACM, pp. 52–60.

[84] Jain, S., Shafique, F., Djeric, V., and Goel, A. Application-level isolation and recovery with solitude. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 95–107.

[85] Jansen, B., Ramasamy, H., Schunter, M., and Tanner, A. Architecting Dependable and Secure Systems Using Virtualization. In *Architecting Dependable Systems V*, R. de Lemos, F. Di Giandomenico, C. Gacek, H. Muccini, and M. Vieira, Eds., vol. 5135 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 124–149.

[86] Jean, E., Jiao, Y., Hurson, A., and Potok, T. Boosting-Based Distributed and Adaptive Security-Monitoring through Agent Collaboration. In *Web Intelligence and Intelligent Agent Technology Workshops, 2007 IEEE/WIC/ACM International Conferences on* (2007), pp. 516–520.

[87] Kephart, J., and Chess, D. The vision of autonomic computing. *Computer 36*, 1 (Jan. 2003), 41 – 50.

[88] Kephart, J. O., Sorkin, G. B., Swimmer, M., and White, S. R. Blueprint for a computer immune system. In *Proceedings of the Virus Bulletin International Conference, San Francisco, California* (1997).

[89] Khan, L., Awad, M., and Thuraisingham, B. A new intrusion detection system using support vector machines and hierarchical clustering. *The VLDB JournalThe International Journal on Very Large Data Bases 16*, 4 (2007), 507–521.

[90] Kitchenham, B. Procedures for performing systematic reviews. *Keele, UK, Keele University 33* (2004), 2004.

[91] Konrad, S., Cheng, B. H. C., Campbell, L. A., and Wassermann, R. Using security patterns to model and analyze security requirements. *Requirements Engineering for High Assurance Systems (RHAS'03)* (2003), 11.

[92] Kramer, J., and Magee, J. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering 16*, 11 (Nov. 1990), 1293–1306.

[93] Kramer, J., and Magee, J. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering, 2007. FOSE '07* (May 2007), pp. 259 –268.

[94] Krka, I., Brun, Y., and Medvidovic, N. Automatic mining of specifications from invocation traces and method invariants. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Hong Kong, China* (2014).

[95] Kumar, G., Kumar, K., and Sachdeva, M. The use of artificial intelligence based techniques for intrusion detection: a review. *Artificial Intelligence Review 34*, 4 (2010), 369–387.

[96] Kwitt, R., and Hofmann, U. Robust Methods for Unsupervised PCA-based Anomaly Detection. *Proc. of IEEE/IST WorNshop on Monitoring, AttacN Detection and Mitigation* (2006), 1–3.

[97] Kwon, M., Jeong, K., and Lee, H. PROBE: A Process Behavior-Based Host Intrusion Prevention System. In *Information Security Practice and Experience*, L. Chen, Y. Mu, and W. Susilo, Eds., no. 4991 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2008, pp. 203–217.

[98] Lane, T., and Brodley, C. E. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conference* (1997), vol. 377, Baltimore, USA, pp. 366–380.

[99] Langner, R. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security Privacy 9*, 3 (June 2011), 49 –51.

[100] Lee, W., Miller, M., Stolfo, S. J., Fan, W., and Zadok, E. Toward cost-sensitive modeling for intrusion detection and response. *Journal of Computer Security 10* (2002), 2002.

[101] Lee, W., Stolfo, S., and Mok, K. A data mining framework for building intrusion detection models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999* (1999), pp. 120–132.

[102] Lee, W., Stolfo, S. J., and Chan, P. K. Learning patterns from unix process execution traces for intrusion detection. In *In AAAI Workshop on AI Approaches to Fraud Detection and Risk Management* (1997), AAAI Press, pp. 50–56.

[103] Lee, W., Stolfo, S. J., and Mok, K. W. Adaptive Intrusion Detection: A Data Mining Approach. *Artificial Intelligence Review 14*, 6 (Dec. 2000), 533–567.

[104] Lemos, R., Giese, H., Mller, H. A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N. M., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Gschka, K. M., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezz, M., Prehofer, C., Schfer, W., Schlichting, R., Smith, D. B., Sousa, J. P., Tahvildari, L., Wong, K., and Wuttke, J. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, R. Lemos, H. Giese, H. A. Mller, and M. Shaw, Eds., vol. 7475. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1–32.

[105] Li, M., and Li, M. An adaptive approach for defending against DDoS attacks. *Mathematical Problems in Engineering 2010* (2010).

[106] Li, X., Zhang, Y., and Li, X. Local Area Network Anomaly Detection Using Association Rules Mining. In *5th International Conference on Wireless Communications, Networking and Mobile Computing, 2009. WiCom '09* (Sept. 2009), pp. 1–5.

[107] Liang, Z., and Sekar, R. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security* (2005), pp. 213–222.

[108] Lo, D., Cheng, H., Han, J., Khoo, S.-C., and Sun, C. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2009), KDD '09, ACM, pp. 557–566.

[109] Lo, D., Mariani, L., and Pezz, M. Automatic steering of behavioral model inference. In *FSE* (2009), ACM, pp. 345–354.

[110] Lorenzoli, D., Mariani, L., and Pezze, M. Towards Self-Protecting Enterprise Applications. In *The 18th IEEE International Symposium on Software Reliability, 2007. ISSRE '07* (Nov. 2007), pp. 39 –48.

[111] Malek, S., Esfahani, N., Menasce, D., Sousa, J., and Gomaa, H. Self-Architecting Software SYstems (SASSY) from QoS-annotated activity models. In *ICSE Workshop on Principles of Engineering Service Oriented Systems, 2009. PESOS 2009* (May 2009), pp. 62 –69.

[112] Malek, S., Medvidovic, N., and Mikic-Rakic, M. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Trans. Softw. Eng. 38*, 1 (Feb. 2012), 73–100.

[113] Malek, S., Mikic-Rakic, M., and Medvidovic, N. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering 31*, 3 (Mar. 2005), 256–272.

[114] Malek, S., Seo, C., Ravula, S., Petrus, B., and Medvidovic, N. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *Int'l Conf. on Software Engineering* (Minneapolis, Minnesota, May 2007), pp. 591–601.

[115] Mannila, H., Toivonen, H., and Verkamo, A. I. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery 1*, 3 (Sept. 1997), 259–289.

[116] Maximilien, E. M., and Singh, M. P. Toward autonomic web services trust and selection. In *Proceedings of the 2nd international conference on Service oriented computing* (New York, NY, USA, 2004), ICSOC '04, ACM, pp. 212–221.

[117] McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. C. A Taxonomy of Compositional Adaptation. Tech. rep., 2004.

[118] Menasce, D., Gomaa, H., Malek, S., and Sousa, J. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *IEEE Software 28*, 6 (Dec. 2011), 78 –85.

[119] Montangero, C., and Semini, L. Formalizing an Adaptive Security Infrastructure in Mobadtl. *FCS'04* (2004), 301.

[120] Morin, B., Mouelhi, T., Fleurey, F., Le Traon, Y., Barais, O., and Jzquel, J.-M. Security-driven model-based dynamic adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2010), ASE '10, ACM, pp. 205–214.

[121] Motahari-Nezhad, H. R., Saint-Paul, R., Casati, F., and Benatallah, B. Event Correlation for Process Discovery from Web Service Interaction Logs. *The VLDB Journal 20*, 3 (June 2011), 417–444.

[122] Mouelhi, T., Fleurey, F., Baudry, B., and Le Traon, Y. A Model-Based Framework for Security Policy Specification, Deployment and Testing. In *Model Driven Engineering Languages and Systems*, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Vlter, Eds., vol. 5301 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 537–552.

[123] Musman, S., and Flesher, P. System or security managers adaptive response tool. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings* (2000), vol. 2, pp. 56 –68 vol.2.

[124] Nagarajan, A., Nguyen, Q., Banks, R., and Sood, A. Combining intrusion detection and recovery for enhancing system dependability. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)* (June 2011), pp. 25 –30.

[125] Neumann, P. G., and Porras, P. A. Experience with EMERALD to Date. In *In 1st USENIX Workshop on Intrusion Detection and Network Monitoring* (1999), pp. 73–80.

[126] Nguyen, Q., and Sood, A. A Comparison of Intrusion-Tolerant System Architectures. *IEEE Security Privacy 9*, 4 (Aug. 2011), 24 –31.

[127] North, D. W. A tutorial introduction to decision theory. *IEEE Transactions on Systems Science and Cybernetics 4*, 3 (1968), 200–210.

[128] Okhravi, H., Comella, A., Robinson, E., and Haines, J. Creating a cyber moving target for critical infrastructure applications using platform diversity. *International Journal of Critical Infrastructure Protection 5*, 1 (Mar. 2012), 30–39.

[129] Okhravi, H., Robinson, E. I., Yannalfo, S., Michaleas, P. W., Haines, J., and Comella, A. TALENT: Dynamic Platform Heterogeneity for Cyber Survivability of Mission Critical Applications.

[130] Ostrovsky, R., and Yung, M. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1991), PODC '91, ACM, pp. 51–59.

[131] Otey, M., Parthasarathy, S., Ghoting, A., Li, G., Narravula, S., and Panda, D. Towards NIC-based Intrusion Detection. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2003), KDD '03, ACM, pp. 723–728.

[132] OWASP.org. Cross-site Scripting (XSS) - OWASP, 2013.

[133] OWASP.org. OWASP Top Ten Project, 2013.

[134] Pal, P., Webber, F., and Schantz, R. The DPASA survivable JBI-a high-water mark in intrusion-tolerant systems. *WRAITS 2007* (2007).

[135] Parveen, P., Weger, Z. R., Thuraisingham, B., Hamlen, K., and Khan, L. Supervised learning for insider threat detection using stream mining. In *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on* (2011), IEEE, pp. 1032–1039.

[136] Pasquale, L., Salehie, M., Ali, R., Omoronyia, I., and Nuseibeh, B. On the role of primary and secondary assets in adaptive security: An application in smart grids. In *2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2012), pp. 165–170.

[137] Perrin, C. The CIA Triad, June 2008.

[138] Poladian, V., Sousa, J. P., Garlan, D., and Shaw, M. Dynamic configuration of resource-aware services. In *Int'l Conf. on Software Engineering* (Scotland, UK, May 2004), pp. 604–613.

[139] Porras, P. A., and Neumann, P. G. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *In Proceedings of the 20th National Information Systems Security Conference* (1997), pp. 353–365.

[140] Portnoy, L., Eskin, E., and Stolfo, S. Intrusion detection with unlabeled data using clustering. In *In Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA-2001* (2001), pp. 5–8.

[141] Portokalidis, G., and Bos, H. SweetBait: Zero-hour worm detection and containment using low- and high-interaction honeypots. *Computer Networks 51*, 5 (Apr. 2007), 1256–1274.

[142] Psaier, H., and Dustdar, S. A survey on self-healing systems: approaches and systems. *Computing 91*, 1 (2011), 43–73.

[143] PuppetLabs. Puppet software, Jan. 2015.

[144] Qin, M., and Hwang, K. Frequent episode rules for Internet anomaly detection. In *Third IEEE International Symposium on Network Computing and Applications, 2004. (NCA 2004). Proceedings* (Aug. 2004), pp. 161–168.

[145] Raissi, J. Dynamic Selection of Optimal Cryptographic Algorithms in a Runtime Environment. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on* (2006), pp. 184–191.

[146] Rawat, S., and Saxena, A. Danger theory based SYN flood attack detection in autonomic network. In *Proceedings of the 2nd international conference on Security of information and networks* (New York, NY, USA, 2009), SIN '09, ACM, pp. 213–218.

[147] Reiser, H., and Kapitza, R. Hypervisor-Based Efficient Proactive Recovery. In *26th IEEE International Symposium on Reliable Distributed Systems, 2007. SRDS 2007* (Oct. 2007), pp. 83 –92.

[148] Reynolds, J., Just, J., Clough, L., and Maglich, R. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences, 2003* (Jan. 2003), p. 8 pp.

[149] Reynolds, J., Just, J., Lawson, E., Clough, L., Maglich, R., and Levitt, K. The design and implementation of an intrusion tolerant system. In *International Conference on Dependable Systems and Networks, 2002. DSN 2002. Proceedings* (2002), pp. 285 – 290.

[150] Sadjadi, S. M. A Survey of Adaptive Middleware. Tech. Rep. Michigan State University Report MSU-CSE-03-35, 2003.

[151] Salehie, M., Pasquale, L., Omoronyia, I., Ali, R., and Nuseibeh, B. Requirements-driven adaptive security: Protecting variable assets at runtime. In *Requirements Engineering Conference (RE), 2012 20th IEEE International* (Sept. 2012), pp. 111–120.

[152] Salehie, M., and Tahvildari, L. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst. 4*, 2 (May 2009), 14:1–14:42.

[153] Savola, R. M., and Heinonen, P. Security-measurability-enhancing mechanisms for a distributed adaptive security monitoring system. In *Emerging Security Information Systems and Technologies (SECURWARE), 2010 Fourth International Conference on* (2010), pp. 25–34.

[154] Saxena, A., Lacoste, M., Jarboui, T., Lcking, U., and Steinke, B. A Software Framework for Autonomic Security in Pervasive Environments. In *Information Systems Security*, P. McDaniel and S. Gupta, Eds., vol. 4812 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007, pp. 91–109.

[155] Schneider, F. Enforceable security policies. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]* (2003), pp. 117 – 137.

[156] Schultz, M., Eskin, E., Zadok, E., and Stolfo, S. Data mining methods for detection of new malicious executables. In *2001 IEEE Symposium on Security and Privacy, 2001. S P 2001. Proceedings* (2001), pp. 38–49.

[157] Shameli-Sendi, A., Ezzati-Jivan, N., Jabbarifar, M., and Dagenais, M. Intrusion response systems: survey and taxonomy. *Int J Comput Sci Network Secur (IJCSNS). v12 i1* (2012), 1–14.

[158] Sibai, F., and Menasce, D. Defeating the insider threat via autonomic network capabilities. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS)* (Jan. 2011), pp. 1 –10.

[159] Sibai, F. M., and Menasce, D. A. Countering Network-Centric Insider Threats through Self-Protective Autonomic Rule Generation. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on* (2012), pp. 273–282.

[160] Song, X., Wu, M., Jermaine, C., and Ranka, S. Conditional Anomaly Detection. *IEEE Transactions on Knowledge and Data Engineering 19*, 5 (May 2007), 631–645.

[161] Sousa, P., Bessani, A., Correia, M., Neves, N., and Verissimo, P. Resilient Intrusion Tolerance through Proactive and Reactive Recovery. In *13th Pacific Rim International Symposium on Dependable Computing, 2007. PRDC 2007* (Dec. 2007), pp. 373 –380.

[162] Sousa, P., Neves, N., Verissimo, P., and Sanders, W. Proactive Resilience Revisited: The Delicate Balance Between Resisting Intrusions and Remaining Available. In *25th IEEE Symposium on Reliable Distributed Systems, 2006. SRDS '06* (Oct. 2006), pp. 71 –82.

[163] Spanoudakis, G., Kloukinas, C., and Androutsopoulos, K. Towards security monitoring patterns. In *Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 1518–1525.

[164] Srikant, R., and Agrawal, R. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology EDBT '96*, no. 1057. Springer Berlin Heidelberg, Mar. 1996, pp. 1–17.

[165] Stakhanova, N., Basu, S., and Wong, J. A Cost-Sensitive Model for Preemptive Intrusion Response Systems. In *21st International Conference on Advanced Information Networking and Applications, 2007. AINA '07* (May 2007), pp. 428 –435.

[166] Stakhanova, N., Basu, S., and Wong, J. A taxonomy of intrusion response systems. *International Journal of Information and Computer Security 1*, 1 (Jan. 2007), 169–184.

[167] Strasburg, C., Stakhanova, N., Basu, S., and Wong, J. A Framework for Cost Sensitive Assessment of Intrusion Response Selection. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International* (July 2009), vol. 1, pp. 355 –360.

[168] Sundaram, A. An introduction to intrusion detection. *Crossroads 2*, 4 (Apr. 1996), 3–7.

[169] Swiderski, F., and Snyder, W. *Threat Modeling*. Microsoft Press, Redmond, WA, USA, 2004.

[170] Swimmer, M. Using the danger model of immune systems for distributed defense in modern data networks. *Computer Networks 51*, 5 (Apr. 2007), 1315–1333.

[171] Taddeo, A. V., and Ferrante, A. Run-time selection of security algorithms for networked devices. In *Proceedings of the 5th ACM symposium on QoS and security for wireless and mobile networks* (New York, NY, USA, 2009), Q2SWinet '09, ACM, pp. 92–96.

[172] Tan, P.-N., Steinbach, M., and Kumar, V. *Introduction to data mining*. Addison Wesley, 2005.

[173] Tang, C., and Yu, S. A Dynamic and Self-Adaptive Network Security Policy Realization Mechanism. In *Network and Parallel Computing, 2008. NPC 2008. IFIP International Conference on* (2008), pp. 88–95.

[174] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[175] The MITRE Corporation. CWE - 2011 CWE/SANS Top 25 Most Dangerous Software Errors, 2011.

[176] The MITRE Corporation. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'), 2013.

[177] The NTP Public Services Project. *The NTP FAQ*.

[178] Uribe, T. E., and Cheung, S. Automatic analysis of firewall and network intrusion detection system configurations. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering* (New York, NY, USA, 2004), FMSE '04, ACM, pp. 66–74.

[179] Valdes, A., Almgren, M., Cheung, S., Deswarte, Y., Dutertre, B., Levy, J., Sadi, H., Stavridou, V., and Uribe, T. An Architecture for an Adaptive Intrusion-Tolerant Server. In *Security Protocols*, B. Christianson, B. Crispo, J. Malcolm, and M. Roe, Eds., vol. 2845 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 569–574.

[180] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering 33*, 12 (Dec. 2007), 856–868.

[181] Venter, J. C., et al. The Sequence of the Human Genome. *Science 291*, 5507 (Feb. 2001), 1304–1351.

[182] Verissimo, P., Neves, N., Cachin, C., Poritz, J., Powell, D., Deswarte, Y., Stroud, R., and Welch, I. Intrusion-tolerant middleware: the road to automatic security. *IEEE Security Privacy 4*, 4 (Aug. 2006), 54 –62.

[183] Vikram, K., Prateek, A., and Livshits, B. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 173–186.

[184] Villegas, N. M., Mller, H. A., Tamura, G., Duchien, L., and Casallas, R. A framework for evaluating quality-driven self-adaptive software systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (New York, NY, USA, 2011), SEAMS '11, ACM, pp. 80–89.

[185] Villegas, N. M., Tamura, G., Mller, H. A., Duchien, L., and Casallas, R. DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive*

Systems II, R. d. Lemos, H. Giese, H. A. Mller, and M. Shaw, Eds., no. 7475 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 265–293.

[186] Wagner, D., and Dean, D. Intrusion detection via static analysis. In *2001 IEEE Symposium on Security and Privacy, 2001. S P 2001. Proceedings* (2001), pp. 156–168.

[187] Wang, F., Jou, F., Gong, F., Sargor, C., Goseva-Popstojanova, K., and Trivedi, K. SITAR: a scalable intrusion-tolerant architecture for distributed services. In *Foundations of Intrusion Tolerant Systems, 2003* (2003), pp. 359–367.

[188] Wang, H., Dong, X., and Wang, H. A Method for Software Security Growth Based on the Real-Time Monitor Algorithm and Software Hot-Swapping. In *Dependable, Autonomic and Secure Computing, 2009. DASC '09. Eighth IEEE International Conference on* (2009), pp. 137–142.

[189] Warrender, C., Forrest, S., and Pearlmutter, B. Detecting intrusions using system calls: Alternative data models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on* (1999), IEEE, pp. 133–145.

[190] Wen, L., Wang, J., Aalst, W. M. P. v. d., Huang, B., and Sun, J. A novel approach for process mining based on event types. *Journal of Intelligent Information Systems 32*, 2 (Jan. 2008), 163–190.

[191] Weyns, D., Malek, S., and Andersson, J. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst. 7*, 1 (May 2012), 8:1–8:61.

[192] White, S. R., Swimmer, M., Pring, E. J., Arnold, W. C., Chess, D. M., and Morar, J. F. Anatomy of a commercial-grade immune system. *IBM Research White Paper* (1999).

[193] Wikipedia.org. 68-95-99.7 rule. https://en.wikipedia.org/wiki/68-95-99.7 rule.

[194] Wu, Y.-S., Foo, B., Mao, Y.-C., Bagchi, S., and Spafford, E. H. Automated adaptive intrusion containment in systems of interacting services. *Computer Networks 51*, 5 (Apr. 2007), 1334–1360.

[195] Xiao, L. An adaptive security model using agent-oriented MDA. *Information and Software Technology 51*, 5 (May 2008), 933–955.

[196] Xiao, L., Peet, A., Lewis, P., Dashmapatra, S., Saez, C., Croitoru, M., Vicente, J., Gonzalez-Velez, H., and Lluch i Ariet, M. An Adaptive Security Model for Multi-agent Systems and Application to a Clinical Trials Environment. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International* (2007), vol. 2, pp. 261–268.

[197] Xie, T., Thummalapenta, S., Lo, D., and Liu, C. Data Mining for Software Engineering. *Computer 42*, 8 (Aug. 2009), 55–62.

[198] Yau, S. S., Yao, Y., and Yan, M. Development and runtime support for situation-aware security in autonomic computing. In *Proceedings of the Third international conference on Autonomic and Trusted Computing* (Berlin, Heidelberg, 2006), ATC'06, Springer-Verlag, pp. 173–182.

[199] Yoshioka, N., Washizaki, H., and Maruyama, K. A survey on security patterns. *Progress in Informatics 5*, 5 (2008), 35–47.

[200] Yu, Z., Tsai, J., and Weigert, T. An Automatically Tuning Intrusion Detection System. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics 37*, 2 (Apr. 2007), 373 –384.

[201] Yu, Z., Tsai, J. J. P., and Weigert, T. An adaptive automatically tuning intrusion detection system. *ACM Trans. Auton. Adapt. Syst. 3*, 3 (Aug. 2008), 10:1–10:25.

[202] Yuan, E., Esfahani, N., and Malek, S. Automated Mining of Software Component Interactions for Self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (New York, NY, USA, 2014), SEAMS 2014, ACM, pp. 27–36.

[203] Yuan, E., Esfahani, N., and Malek, S. A Systematic Survey of Self-Protecting Software Systems. *ACM Trans. Auton. Adapt. Syst. 8*, 4 (Jan. 2014), 17:1–17:41.

[204] Yuan, E., and Malek, S. A taxonomy and survey of self-protecting software systems. In *2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (June 2012), pp. 109 –118.

[205] Yuan, E., and Malek, S. Mining software component interactions to detect security threats at the architectural level. In *Proceedings of the 13th Working IEEE/IEIP Conference on Software Architecture* (Venice, Italy, Apr. 2016), WICSA 2016.

[206] Yuan, E., Malek, S., Schmerl, B., Garlan, D., and Gennari, J. Architecture-based Self-protecting Software Systems. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures* (New York, NY, USA, 2013), QoSA '13, ACM, pp. 33–42.

[207] Zhu, M., Yu, M., Xia, M., Li, B., Yu, P., Gao, S., Qi, Z., Liu, L., Chen, Y., and Guan, H. VASP: virtualization assisted security monitor for cross-platform protection. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (New York, NY, USA, 2011), SAC '11, ACM, pp. 554–559.

Curriculum Vitae

Mr. Yuan has over 20 years of professional experience in information technology and management consulting in both commercial and public sectors. In numerous customer engagements, he has played key roles in IT strategy, technical leadership, program management, software architecture and design, COTS integration, and systems engineering to deliver complex, mission-critical enterprise solutions. His in-depth expertise and researh interests cover software architecture, software engineering, autonomic and self-adpative systems, cloud computing, cyber security, software dependability, machine learning, and data analytics.

Mr. Yuan is currently a program director at a Federally Funded Research and Decelopment Center (FFRDC) providing engineering advisory and program oversight to government customers. Prior to that, Mr. Yuan was a program manager and senior architect in Booz Allen Hamilton providing strategy and technical leadership for a number of transformational initiatives for federal and military customers. He also helped build the firm's intellectual capital on SOA and Mission Engineering offerings. His earlier experience in the private sector included E-Commerce technologies, Business-to-Business solutions, and telecommunications systems in Fortune 500 companies.

Mr. Yuan received his Master of Science degree in Systems Engineering from University of Virginia in 1996, and his Bachelor of Science dual degree in Management and Computer Science from Tsinghua University, Beijing in 1993.