Quantitative Security Analysis for Service-Oriented Software Architectures

By

Michael Yanguo Liu

M.A.Sc., University of Victoria, 2003
B.Eng, Harbin Institute of Technology, 1999

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

Supervisory Committee

By

Michael Yanguo Liu

M.A.Sc., University of Victoria, 2003
B.Eng, Harbin Institute of Technology, 1999

Supervisory Committee

Dr. Issa Traore
Supervisor

Dr. Kin Fun Li
Departmental Member

Dr. Stephen W. Neville
Departmental Member

Weber-Jahnke, J.H.
Outside Member

Dr. John Mullins
Additional Member

ABSTRACT

Supervisory Committee

Dr. Issa Traore
Supervisor
Dr. Kin Fun Li
Departmental Member
Dr. Stephen W. Neville
Departmental Member
Weber-Jahnke, J.H.
Outside Member
Dr. John Mullins
Additional Member

Due to the dramatic increase in intrusion activities, the definition and evaluation of software security requirements have become important aspects of the development of software services. It is now a well-accepted fact in software engineering that security concerns, like any other quality concerns, should be dealt with in the early stages of software development process. Current practices for software security architecture risk analysis, however, still heavily rely on human expertise. This involves a significant amount of subjective efforts creating a greater potential for inaccuracies. In this dissertation, we propose a framework for quantitative security architecture analysis for service-oriented software systems. In this regard two important contributions are made in the dissertation. First, we identify and define some internal security attributes and related properties based on a generic service-oriented software model, setting up a framework for the definition and formal evaluation of corresponding security metrics. Second, we propose a measurement abstraction paradigm named User System Interaction Effect (USIE) model that can be used to systematically derive and analyze security concerns from service-oriented software architectures. Many aspects of the model derivation and analysis can be automated, which limit the amount of user involvement and, thereby, reduce the subjectivity underlying typical security analysis process. The model can be used as a foundation for quantitative analysis of software services from different security perspectives with respect to the internal security properties introduced. Based on sample metrics derived from the framework, we illustrate empirically the viability of our paradigm by conducting case studies based on existing open source software.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

SOA	Service-Oriented Architecture
SOAD	Service-Oriented Architecture Development
USIE	User System Interaction Effect
ASA	Attack Surface Analysis
TAM	Threat Analysis and Modeling
CERT	Computer Emergency Response Team
SDL	Security Development Lifecycle
TCSEC	Trusted Computer System Evaluation Criteria
ITSEC	Information Technology Security Evaluation Criteria
SM	Security Measurement
SAAM	Software Architecture Analysis Method
UML	Unified Modeling Language
SAM	Software Architecture Model
COD	Component Oriented Design
OOD	Object Oriented Design
OO	Object Oriented
DOS	Denial of Service
STEM	Security Testing and Engineering using Metrics
XMI	XML Metadata Interchange

Acknowledgments

Firstly, I would like to express my deepest gratitude to my supervisor, Dr. Issa Traore. He is always responsible and supportive for my graduate study. It has been great pleasure to work with him on the academic research. I truly appreciate his help, encouragement and financial support.

Secondly, I would like to thank Dr. Kin F. Li, Dr. Stephen W. Neville, Dr. Jens H. Weber and Dr. John Mullins for being my PhD committee members and providing valuable comments to this dissertation.

Thirdly, I want to thank my colleagues in the ISOT group for their enlightening discussion and priceless friendship. Particularly, I want to express my appreciation to Mr. Akif Nazar, who contributed to the implementation of STEM toolkit.

Fourthly, I wish to thank our staff Ms. Vicky Smith, Ms. Moneca Bracken and Ms. Mary-Anne Teo for their kind support during my graduate study.

Finally, my special thanks go to my family for their deep love and strong support on the pursuit of my Ph.D. degree.

Chapter 1

Introduction

1.1 Context

It is commonly agreed that software carries the biggest security challenges of today's systems. According to [68], about 20 new software vulnerabilities are reported weekly. This situation has increased security awareness in the software community. Today, software services are expected not only to satisfy functional requirements but also to comply with security requirements. As demand for more trustworthy systems is increasing, the software industry is also adjusting itself to security standards and practices by increasing security assessment and testing efforts. However, unlike software quality attributes such as maintainability, reliability and performance that have been widely researched over decades, the study of software security still remains immature due to its complex and multifaceted nature.

Traditional approach to software security engineering referred to as "penetrate and patch" consists of fixing security flaws after they have become known [25]. "Penetrate and patch" is a fictitious solution which deals only with the symptoms and not the deep causes of the problem. In addition, this approach has proven to be inadequate as an engineering approach since it usually uncovers problems only after the software system has been fully developed and delivered. There is a consensus that better software engineering requires improving software quality in the early stage of software development, preferably at the architecture level.

Recently, service-oriented modeling has emerged as an effective technique for specifying and designing software architectures. Service-Oriented Architecture (SOA) lays the foundation of a new distributed framework that facilitates exposure of software

components as services. In reality, many security exploits in software systems are caused by malicious uses of publicly available services [73]. Accordingly, it is essential to guarantee a high level of security in the design and implementation of software services. Even though several approaches and notations have been proposed to model security concerns under the SOA development framework [34], a huge amount of subjective and manual effort and expertise is still required in the security assessment and evaluation of software services designs. Likewise the best modeling techniques cannot prevent security flaws from slipping through design models.

As noted by Meyer, software engineering is the process of producing quality software [54]. So quality is the driving force of the engineering aspect. Software quality consists of the combination of several attributes also referred to as quality attributes or quality factors in the literature. There are two kinds of quality attributes: external and internal attributes. External attributes refer to software qualities whose presence or absence may be detected by the stakeholders (e.g., users, customers, developers etc.). Examples of such attributes include reliability, maintainability, efficiency, compatibility, portability and so on. Internal attributes correspond to hidden software qualities meaningful only to software professionals (e.g., developers, analysts, testers) who have access to software work products such as code, specification, or documentation. Examples of internal software attributes include readability, complexity, and modularity. Ultimately what matters from the perspective of the users or the customers are the external attributes for which they have a clear perception. External attributes, however, can only be achieved by judiciously applying techniques internal to software construction ensuring internal qualities. As argued by Meyer, “the internal techniques are not an end in themselves, but a means to reach external software qualities” [54].

Security, as an external attribute, is a multifaceted quality concept, in which each facet represents a separate external software attribute in its own [58]. Traditionally research in software security has focused primarily on analyzing external security attributes such as confidentiality, integrity, and availability. Identifying breaches in these attributes can be quite difficult and costly. In this context, the security community is moving progressively towards easily interpretable security attributes. One such attribute that is widely used in the research literature and in the industry is software vulnerability.

Vulnerability in software corresponds to a software flaw that can be exploited to conduct an attack against the software system. Analogy can be established between vulnerabilities and defects. As a matter of fact the most common metric of vulnerability currently in use is *vulnerability count* similar to *defect count* for functional flaws. The analogy goes far beyond that: vulnerability count is commonly used as a predictor of software security the same way defect count has been used for a while as a predictor of software reliability. Fenton and Neil, however, in their critique of defect prediction models, highlight the weakness of defect count as a metric of reliability [21]. One of the main reasons they put forward being that it is extremely difficult to predict how defects discovered during testing or inspection will manifest when the system operates in practice. First, there is a great variability among defects in terms of their seriousness. Some defects will have significant impact on the system and its users when they result in failures, while others will lead to very minor consequences. Second, there is a great variability in the likelihood of defects resulting in failures; in fact only a very limited percentage of all defects actually results in noticeable failures. Fenton and Neil conclude that using defect count as overall quality predictor is misleading. The same remark applies to software vulnerabilities since there is a great variability in the likelihood of a vulnerability resulting in a successful attack. In case where some vulnerabilities result in successful attacks, there is a great variability in their seriousness. In this context, an alternative software security attribute, which is more and more gaining in interest, has been termed in the literature as software *attackability*. Attackability is a concept proposed by Howard and colleagues to measure the extent that a software system or services can be the target of successful attacks [28]. They define *attackability* as the short word for attack opportunity. More specifically, attackability is the likelihood that a successful attack will happen on a software system or some specific services. Attackability can be further specified with respect to particular software attacks. For instance, Denial of Service (DOS) attackability refers to the likelihood that a successful DOS attack will happen.

1.2 Research Problem

Even though the concept of attackability provides a simple and practical foundation for software security analysis, how to analyze and mitigate effectively attackability for software products during the design process still remains unaddressed. So far attackability is currently addressed in the software industry through patching. This corresponds to the so-called "penetrate-and-patch" approach, which, as argued above, is unsatisfactory. A more effective solution is to address attackability concerns before the delivery of the final software product. So, attackability analysis should be part of the software development process.

In order to improve the security of software in early development stages, we need to identify some internal attributes, which may influence directly or indirectly software security qualities. So far, a limited amount of work has been done on security analysis at the software architecture level. Most of the published works are based on formal methods [35], [66], [16], which due to their esoteric nature face a huge barrier for their adoption by the industry. In this context, software metrics systems could represent a more practical alternative for software security analysis. Generally, software metric is a measure of some property of a piece of software or related artifacts. Security metrics are needed to understand current state-of-security, to improve that state, and to obtain resources for improvements. Vaughn questions the feasibility of "measures and metrics for trusted information systems" [67]. According to him, metrics are possible in disciplines such as mechanical or civil engineering because they comply with the laws of physics, which can be used to validate the metrics. In contrast, the system engineering discipline (and software engineering also) is not compliant with the laws of physics and presents more of a challenge in proving correctness. He points out, however, that there are a host of measures and metrics that may be useful in predicting systems security characteristics including penetration success rates, coupling and cohesion of security relevant software, testing defect rates, process quality and so on. Vaughn, therefore, suggests that we can develop effective metrics for systems security by accepting some risk in our use of metrics and by validating them in the real world through empirical investigation and experimentation.

Software design metrics have been successfully developed for a broad range of quality attributes including reliability, performance, and maintainability [27], [12]. However, the field of software security is still immature [22]. Although extensive works have been achieved on developing measurement properties for (traditional) internal software attributes [19], [20], [39], [52], [65], [72], to our knowledge little attention has been paid to such issue in research on security metrics. Some of the few published works on this issue include [71] where Wang and Wulf motivate the rationale for theoretical validation of security metrics and suggest possible security attributes. However measurement concepts are barely defined or formalized. Based on these considerations, we propose in this work to develop a framework for quantitative analysis of software security at the architectural level with a focus on service oriented architecture (SOA). The proposed framework will attempt to address many of the limitations outlined above concerning architectural-level security analysis. We discuss in the next section our approach to achieve this objective.

1.3 Proposed Approach

Our ultimate goal in this research is to develop a measurement framework that can assist security architects in analyzing systematically and objectively software security qualities at the architectural level. We propose to achieve this objective by relating software attackability with internal software attributes. This will allow analyzing and mitigating software attackability by manipulating the corresponding internal attributes in a systematic way. The analysis will be driven by a family of security metrics which may be organization specific. Figure 1.1 depicts our proposed approach, which involves two perspectives, namely metrics development and metrics application.

The metrics development perspective involves a process and some mechanisms to develop meaningful architecture-level security metrics and related attackability evaluation guidelines. Specifically, our metrics development process follows three key steps. Firstly, a suite of internal security metrics that can be used as predicting factors for software attackability need to be defined using an appropriate security measurement abstraction. In this work, we propose and use a new security measurement abstraction paradigm named

the *User System Interaction Effect (USIE)* model. The metrics definition can be guided appropriately by the internal security attributes and properties identified in our approach. Secondly, the suite of metrics defined must be validated theoretically with respect to the corresponding security measurement attributes and properties. Thirdly, the relationships between the internal security metrics and corresponding attackability metric must be established and validated through empirical studies. The final outcome of the metrics development process is a suite of valid security metrics with their relationships to specific software attackability concepts.

The metrics development perspective is intended for quality assurance (QA) personnel, who need to design a family of security metrics for their organizations. The metrics application is intended for quality management or quality control personnel who use the metrics developed by QA as quality predictors during the design and verification of software products.

The metrics application perspective depicts how the internal security metrics derived from our approach can be used for practical software architecture security evaluation. Specifically, in our approach, the evaluation starts by transforming the system design artifacts into the USIE models. The generated USIE models serve as the basis to compute the measurement data for the predefined security metrics. Then attackability estimations with respect to the provided attack scenarios can be computed indirectly through the predefined attackability evaluation guidelines identified in the metrics development process. The computed metrics allow security analysts to make decisions about the quality of the design. The security evaluation process can proceed iteratively if some modifications or updates are needed to the service designs.

1.4 Contributions

In order to achieve the research goal outlined above, we make four major contributions in this dissertation.

Firstly, we identify a collection of internal software attributes that may have an impact on security. We define an abstract software model for SOA and use this model to define measurement properties for internal security attributes. This provides a basis for

theoretical validation of internal security metrics, which is an important step in establishing the meaningfulness of security metrics. This contribution has been published in [49].

Secondly, we identify and study empirically the relationships between the security-related internal attributes and attackability as external attribute. So far, some of these relationships were assumed and used informally in the literature, but no empirical evidence was provided before our work. This has led to publications, [43], [44], [47], [48].

Thirdly, we propose a measurement abstraction to define and derive security metrics from SOA design models. The proposed paradigm, named User System Interaction Effect (USIE) model, provides a basis for quantitative security analysis at the architecture level. This has given rise to publications [42], [45], [46].

Fourthly, we propose a set of techniques and metrics for analyzing security attributes from different security perspectives. These allow identifying security design flaws and adopting, accordingly, appropriate mitigation solutions. This has been submitted for journal publication as [50].

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 summarizes and discusses related work on secure software development. Chapter 3 presents our first contribution by defining the theoretical foundation for the analysis of internal software security attributes. Chapter 4 introduces a new measurement abstraction for service-oriented architecture, the User System Interaction Effect (USIE) model. Chapter 5 illustrates how to conduct systematic security analysis of service-oriented architecture using the proposed USIE paradigm. Several internal security metrics are also introduced in this chapter. Chapter 6 presents an empirical study based on open source software for the validation of the security design principles and metrics introduced in this work. Finally, Chapter 7 concludes this dissertation and discusses future work.

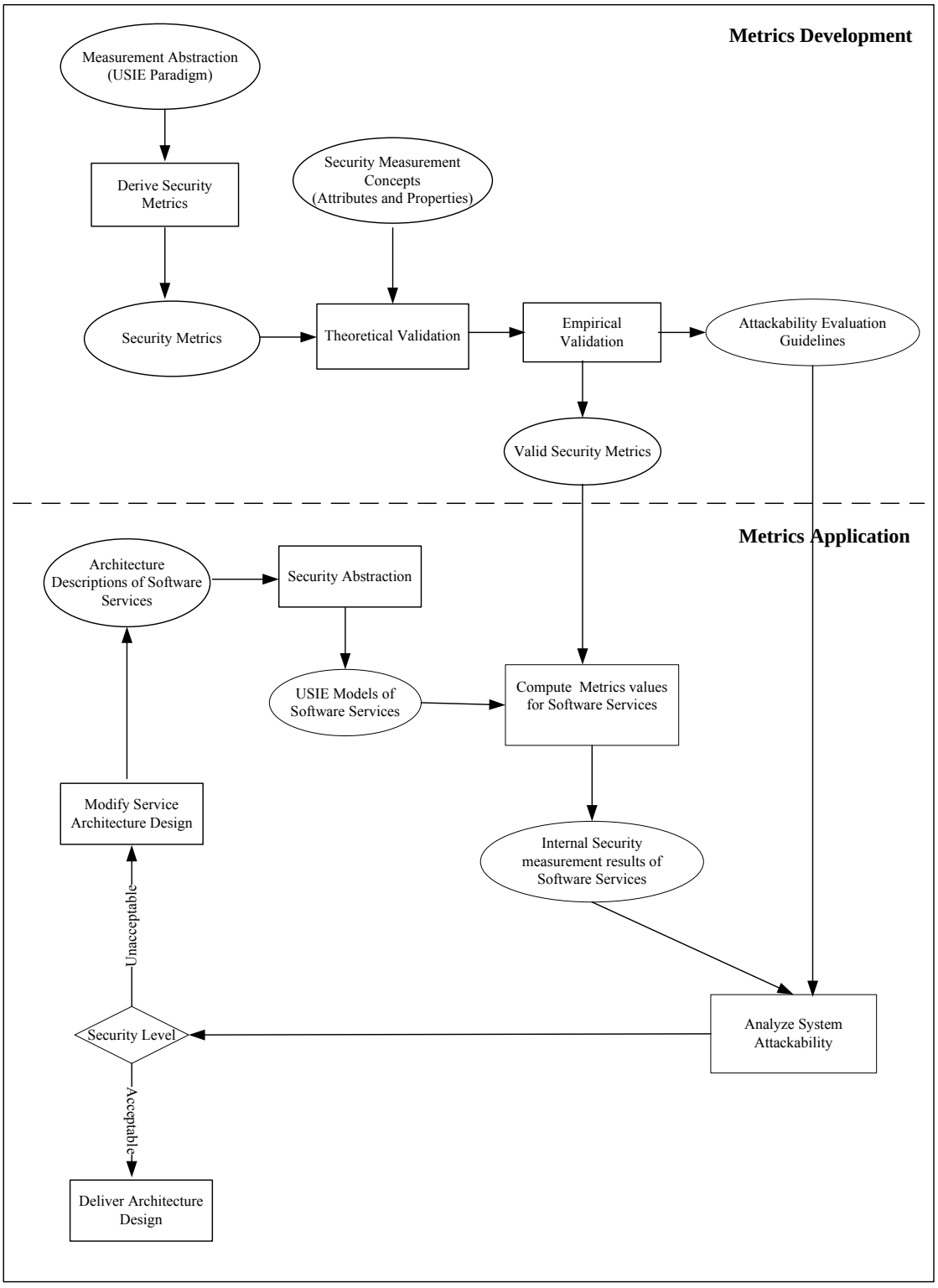

Figure 1.1. Architecture Security Evaluation Framework and Approach

Chapter 2

Literature Review

In this chapter, we summarize and discuss related works. In section 2.1, we review series of works on the definition and evaluation of software metrics, which is the main focus of this dissertation. In section 2.2, we survey specific approaches of security analysis proposed in the literature, and also discuss how some of these approaches are used within the context of sound engineering process. In section 2.3, we survey existing works on secure software development lifecycle and methodologies.

2.1 Design and Validation of Security Metrics

According to the systems security engineering capability maturity model metrics committee (SSE-CMM), "security metrics focus on the actions (and results of those actions) that organizations take to reduce and manage the risks of loss of reputation, theft of information or money, and business discontinuities that arise when security defenses are breached" [33]. One of the earliest workshop on security metrics organized by the National Institute of Standards and Technology (NIST) and the Computer System Security and Privacy Advisory Board (CSSPAB), has highlighted the fact that security metrics represents a challenging discipline, which is still immature [59]. However, some researchers have started devoting themselves to research on security metrics for software design. In this section, we introduce some of the pioneer work in this field.

2.1.1 Security Measurement Framework

In [71], Wang and Wulf motivated the rationale for quantitative evaluation of software security and proposed what they called a security measurement (SM) framework. The aim

of the SM framework is to provide a systematic way to estimate the security strength of a system or a family of closely related systems. Specifically, the SM framework consists of four aspects described as follows:

1) *Definition of Software Security***:** SM framework addresses the definition of software security by using the concept of security attributes. Specifically, the security of a software application is captured by one or more security attributes. Therefore, defining security attributes is system independent.

2) *Selection of Units and Scales:* An attribute can be measured in many different units and scale types. In the SM framework, the attributes identified to interpret security must be assigned appropriate measurement scales and units. Plausibility and accuracy are the two primary concerns in the SM framework when selecting measurement scales and units.

3) *Definition of an estimation methodology***:** SM framework considers software security to be function of a host of attributes and the interactions between these attributes. Specific estimation methods must be defined for the attribute set to approximate the security strength of a software system. SM introduces several candidate estimation methodologies such as simple decomposition, functional relationship and weighting plus prioritizing etc.

4) *Validation of Metrics***:** security metrics developed need to be validated before they can be adopted. The SM framework doesn't define formal guidelines for the validation of security metrics, but presents a few thoughts toward validation of the security measurements. They suggest that security metrics can be validated using three possible methods, namely validation based on measurement theory, validation using case studies, and validation using formal experiments.

In summary, Wang and Wulf's work is one of the earliest research efforts on software security metrics development. However, even though the SM framework defines a high level process, in which sound security metrics can be developed, it still needs to be refined and improved before being applied in practice. As mentioned by Wang and Wulf, they need to continue to engage in efforts to define specific guidelines for the

identification of software security properties, the development of estimation methodologies and measurement validation strategies. Like Wang and Wulf, we propose in this work a framework for developing security metrics. But our work goes beyond their proposal by providing a concrete and practical foundation and related methodologies for metrics development and validation.

2.1.2 Measurement Concepts Definitions

In the last two decades, several efforts have been made towards rigorous definition of software attributes and their metrics. While some of these works emphasize the application of traditional measurement theory [78] to software metrics [20], [39], [52], others focus on formally defining expected properties of software attributes within an axiomatic framework [72], [8], [56].

The first work on axiomatic validation of software metrics, authored by Elaine Weyuker, proposed a set of axioms to serve as a basis for the validation and evaluation of software complexity metrics [72]. Specifically, nine axioms were proposed and formalized in Weyuker's work, each representing an expected property of complexity metrics. As an example, Weyuker's monotonicity axiom states that "the components of a program are no more complex than the program itself". Weyuker claimed that these axioms represent desirable and relevant properties of software complexity, but are certainly not complete. Weyuker also evaluated and compared several known complexity metrics using her axioms to clarify the strengths and weaknesses of the examined complexity metrics. She questioned the usefulness of these metrics in measuring synthetic complexity by showing that none of these metrics possesses all nine properties and several fail to exhibit fundamental properties.

Based on a different perspective, Kitchenham *et al.* proposed a validation framework for software metrics based on measurement theory, and centered on the notion of software entities and software attributes [39]. They defined several criteria for theoretical and empirical validation of software metrics. One of the most important and somewhat controversial of these criteria stated "any definition of an attribute that implies a particular measurement scale is invalid" [39]. As this came as a shortcoming of previous

axiomatic approaches such as [72], it triggered a discussion between the authors and some of the tenants of the axiomatic approaches [40], [56]. From this discussion, we can retain as response to the criticism of the axiomatic approach that excluding any notion of scales in the definition of software metrics will simply lead to abstracting away important relevant information, weakening as a consequence the checking capabilities of corresponding validation framework [56]. To corroborate their claim, they took as an example the case of experimental physics, which "has successfully relied on attributes such as temperature that imply measurement scales in the definition of their properties" [56]. According to them, deriving and associating properties with different measurement scales can define an attribute. In this case, given a software attribute metric, only properties associated with relevant scales would be used to check it. Morasca and Briand refined this perspective by proposing a hierarchical axiomatic approach for the definition of metrics of software attributes at different scales [57]. In their approach, different collections of axioms are associated with a software attribute, each relevant to specific measurement scale. Their work is significant in the sense that it establishes how axiomatic approaches relate to the theory of measurement scales, and also helps addressing consistency issues in the axiom systems.

The above works represent a small but representative amount of the large amount published works on theoretical validation of software metrics. However, none of the existing works have studied measurement properties from a security perspective. The focus of the research has so far been on studying traditional software quality attributes such as reliability and maintainability. To our knowledge, the only attempt to define security measurement properties is Millen's work [55], which will be discussed in the next session.

2.1.3 Survivability Measurement Framework

Survivability is the quantified ability of a system, subsystem, equipment, process, or procedure to continue to function during and after a natural or man-made disturbance. In [55], Millen proposed a theoretical framework for the definition and validation of survivability metrics based on service-oriented architecture.

Millen defined a generic system model based on a hierarchical structure that reflects the engineering depth of real software systems. Specifically, Millen defined a system as "a set of components configured to provide a set of user services". A configuration consists of a collection of components connected in a specific way, each providing specific services. These services are referred to as the supporting services for the corresponding configuration. Based on this specific model, Millen defined several properties that characterize survivability metrics. He also suggested a sample survivability metric and validated the metric using the defined properties.

Millen's work focuses on survivability, which is only one aspect of security [58]. Furthermore, his framework targets system in general and does not really consider the specific characteristics of software systems. Nonetheless, to our knowledge, it is the only attempt to defining measurement properties from a security perspective. In our work, we define a generic service-oriented software model that is an extension of the generic service-oriented model proposed by Millen. Our model goes beyond the original model of Millen by capturing specific characteristics of software systems.

2.2 Security Analysis: Models and Approaches

There has been a great range of work in security analysis of computer system, which includes the areas of adversary modelling, attack specification, vulnerability analysis, security-related taxonomies and databases, etc. In general, security analysis involves the analysis of system threats and vulnerabilities and their potential impact on the system's mission. In this section, we introduce some representative security analysis techniques in the literature.

2.2.1 Attack Surface Analysis

In [28], Howard *et al.* proposed to use *attack surface* to determine whether one version of software application has less attackability than another. They define attack surface in terms of system actions that are externally visible to system's users and the resources

accessed or modified by each action. Intuitively, more explored attack surfaces lead to high likelihood of being successfully attacked.

The Attack Surface Analysis (ASA) was originally inspired by Saltzer and Schroeder's security design principles which were outlined in 1975 for engineering secure software systems [60]. Over decades, security engineers have applied Saltzer and Schroeder's principles to guide the design and implementation of secure computing systems.

The Attack Surface Analysis of a software application is the process of enumerating and reducing all the accessible entries with high likelihood of being attacked. For example, a remotely accessible socket opened by a software application has the potential to be misused by attackers, therefore can be considered to be an attack surface. Figure 2.1 depicts a typical process of ASA. In this process, several security related questions derived from Saltzer and Schroeder's need to be answered and some actions need to be taken accordingly. In summary, security analysts when conducting ASA at design level typically focus on four aspects: 1) Reducing the amount of attack surfaces that are granted by default. 2) Restricting the scope of physical access to the attack surfaces. 3) Restricting the scope of identities that can access the attack surfaces. 4) Reducing the privilege carried by the attack surfaces.

The proposed framework can be used to compare the security level of different versions of a software system. Basically, in the methodology, the number of attack surfaces is used as a metric of system attackability. However, no evidence is provided in [28] that show that the count of attack surfaces correlates with external attackability. Furthermore, defining the classes of attack surfaces is application specific and requires human expertise. As a result, the classes of attack surfaces for a given system may vary from one expert to the other, which limits the objectivity of the methodology.

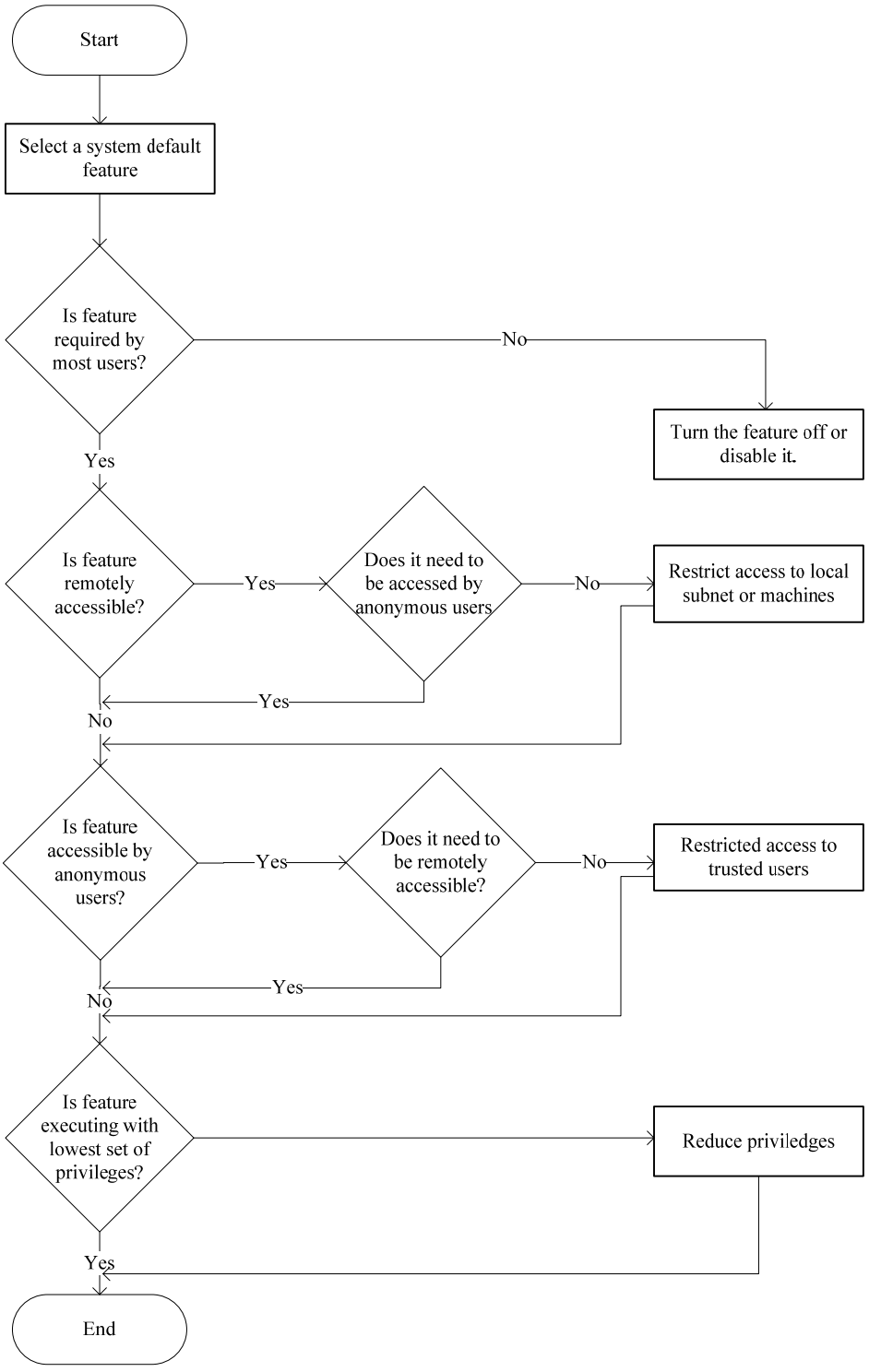

Figure 2.1. Attack Surface Analysis Process

2.2.2 Microsoft Threat Analysis and Modeling Method

Threat Analysis and Modeling (TAM) is a methodology that is used for threat identification and profiling in Microsoft Security Development Lifecycle (SDL) process [24], which is discussed later. TAM is a critical part of the SDL process in the sense that threat profiles, which are the outcome of TAM activities, specify the particular security concerns underlying the application context and direct the design and implementation from security point of view. A threat profile usually contains the descriptions of potential threats, the application vulnerabilities associated to each of the threats, and the impact and probability factors of each potential threat.

TAM processes may be conducted either from attacker's perspective or from system perspective. The attacker-centric approach starts with the identification of potential attacking roles and continues with the definition of corresponding attack profiles, attacking objectives and steps. The system-centric approach starts with understanding the design and implementation of the system and then continues by identifying and profiling attacking entries for the system. From a general perspective, the TAM process involves the following three steps:

1) **Preparation:** at this stage, the security team focuses on collecting necessary information for the core threat-modeling analysis. The documentation and artefact required typically include requirement specifications, software architecture underlying security assumptions, and information on system external dependencies, etc.
2) **Threat Analysis:** at this stage, the security team should uncover and document as much as possible threat scenarios for the application, and also determine the threat type and risk level for each identified threat scenario. For instance, Microsoft uses a threat taxonomy called STRIDE (which is an acronym for Spoofing Identity, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) to identify various threat types. Microsoft defines the threat risk into 4 levels where risk level 1 is the highest and risk level 4 is the lowest.
3) **Threat Mitigation:** This step is performed after establishing threat models. The security team needs to consider the threat model to determine the appropriate

remedies to the threats. Microsoft defined a small set of mitigation strategies that can be described as "*Do nothing", "Turn off Feature", "Remove Feature", "Warn User",* and "*Counter the threat with technology"*. Specific technologies need to be provided if the mitigation strategy is determined to be "*Counter the threat with technology"*.

The TAM process is supported by the Microsoft Application Security Threat Analysis & Modeling tool. This tool provides a user friendly interface for security experts to specify threat scenarios for a software application. It has also the capability to assimilate the information provided to build security artefacts such as access control matrices, data flow and trust flow diagrams, and focused, customizable reports.

2.2.3 Privilege Graph Paradigm

Dacier and Deswartes proposed for computing systems a high level privilege graph that can be used to estimate the possibility of security breaches from a potential attacker [15]. Specifically, a privilege graph is a directed graph in which nodes represent the privileges owned by a user or a group of users, and edges represent potential privilege transfer. Specifically an arc from node n_1 to node n_2 corresponds to an attack method that can allow the owner of the privileges in n_1 to obtain those in n_2. An estimated success rate of the corresponding attacks is also assigned to each of the arcs.

As an example, Figure 2.2 shows a privilege graph where the nodes are labelled by system roles and the arcs are labelled by attack methods. In the privilege graph, some roles can be marked as "attack target" nodes since they carry highly sensitive privileges such as super-user privileges and are most likely targets of attacks. On the other hand, some nodes can be identified as "attacker" nodes if the corresponding role can represent a potential attacker of the system. For instance, in Figure 2.2, we can define the "unregistered user" node to be an attacker node and the "administrator" node to be an "attack target" node. If a path exists between an attacker node and a target node, then a security breach can potentially occur since a possible attacker can exploit system vulnerabilities transitively to obtain the target privileges. The difficulty for an attacker to reach its target can be estimated if each arc in the privilege graph is assigned a weight

corresponding to the "effort" needed for a potential attacker to perform the privilege transfer corresponding to this arc. The "effort" is determined based on security expertise.

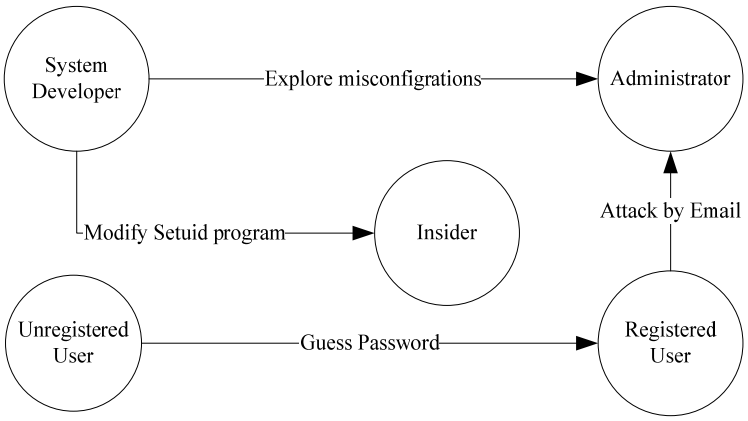

Figure 2.2. Example of Privilege Graph

Privilege graphs can be used to derive attack scenarios describing how intruders misuse available privileges and obtain unauthorized privileges. Even though systematic security analysis can be conducted using privilege graphs, the construction of privilege graphs still requires a certain level of security expertise. Particularly, a certain amount of subjective analysis is involved in estimating quantitatively the "effort" for the arcs of the graphs.

2.2.4 Other Security Analysis Frameworks

In [13], [26], [41], [64], bugs count is used as a metric of software security, where software bugs are collected from either static inspections or testing reports. Using defects count to predict system quality, as discussed in [21], is not reliable. Firstly, static inspections can generate false alarms. The exposure of the defects depends on how the system will operate in practice; some of the defects may never lead to security breaches. Secondly, the effectiveness of dynamic techniques to find defects, such as testing, depends on the quality of the testing process. Different testing efforts may lead to various results. Furthermore, it is difficult to determine in advance the seriousness of a defect; in practice, a very small portion of defects in a system will cause almost all the observed security breaches.

In the area of operational security measurement, Brocklehurst *et al.* [9] studied the ability of a computer system to resist attacks by estimating the attack effort and attack reward. Their approach is based on the analogy between system reliability and security. As time is used to model system reliability (e.g., Mean Time to Failure), they proposed to use attack effort and reward to model system security (e.g., Mean Effort to Security Breach). A probabilistic model for operation security is suggested in their work.

Alves-Foss *et al.* [1] measured computer system vulnerability by evaluating malevolent and neglectful acts affecting system security. They introduced a method, named the System Vulnerability Index (SVI), which uses a number of factors that affect security and a rule-based expert system that evaluates security factors based on a set of rules. The SVI framework gives an indication of the presence of certain conditions that could lead to security breaches, and relies on system administrators to resolve the problems. The SVI framework helps reducing security risks solely in the operational stage of software lifecycle.

Voas *et al.* [69] proposed a metric based approach to assess relative security among different versions of the same software system. Their approach, which is named Adaptive Vulnerability Analysis (AVA), exercises software source code by simulating incoming malicious and non-malicious attacks. A quantitative metric is computed by determining whether the simulated attacks undermine the security requirement of the system. AVA applies fault-injection techniques to the source code of software applications, which makes AVA only applicable in the late stage of software development. In contrast, our framework targets primarily the software design phase.

Most of the works summarized above focus primarily on system-level security or code-level security, without really considering design artefacts, which are the primary target of our work

2.2.5 Security Analysis vs. Security Risk Analysis

Many decision makers of information technology (IT) organizations assume that security analysis is the same thing as security risk analysis. However, these two processes are very different.

Specifically, a good security analysis usually explores the design and implementation of the target system and delivers a comprehensive report that includes detailed information about the exploits and possible threats the system may be vulnerable to. The security analysis is also responsible to rank these exploits and threats according to their risk levels and provide recommendations for mitigating actions. In general, security analysis is conducted after the design or implementation artefacts have been produced.

Security risk analysis, also known as security risk assessment, is a process that an organization goes through to determine their risk exposure. A risk in this context is defined as the possibility that a particular damage could happen to a business or organization and the impact that the damage could cause. The goal of a risk analysis is to integrate financial objectives with security objectives, therefore, the activities of security risk analysis is usually performed before the design stage of the system development.

According to Peltier [80], there are two approaches of security risk analysis, which are respectively quantitative approach and qualitative approach.

Quantitative risk analysis approach employs basically two fundamental elements: the probability of an event occurring and the likely loss should it occur. A figure named "Annual Loss Expectancy (ALE)" is usually calculated for an event by simply multiplying the potential loss by the probability. Accordingly, it is theoretically possible to rank events in order of risk (ALE) and to make decisions based upon this. Although a number of organisations have successfully adopted quantitative risk analysis approach, this type of risk analysis usually suffers from the unreliability and inaccuracy of the data.

Qualitative risk analysis approach is by far the most widely used approach to risk analysis. Instead of assigning numbers and values to components and losses, this approach usually goes through different scenarios of risk possibilities and ranks the seriousness of the threats and validity of the different possible countermeasures. A number of interrelated elements are involved in the qualitative risk analysis process including threats, vulnerability and countermeasures. The final report of a qualitative risk analysis helps the adoption of the best countermeasures against potential system threats.

2.3 Secure Software Development: Lifecycle and Methodologies

In this section, we give some background information on Microsoft security development lifecycle (SDL) and present a brief survey on the approaches and techniques proposed or used in industry and academia for secure software development.

2.3.1 Microsoft SDL

In this section, we give an overview of SDL by covering three different aspects, the origin of SDL, the SDL process and the future evolution of SDL.

2.3.1.1 The Origin of SDL

In the early 1980s, having realized the growing importance of computer security, the United States National Security Agency (NSA) developed a set of evaluation criteria known as the Trusted Computer System Evaluation Criteria (TCSEC) or the "Orange Book". The purpose of these criteria is to identify the required security features and provide general guidelines for security assurance of trusted computer systems including the underlying software systems. The Orange Book defined hierarchically several classes of security evaluation in which higher classes require higher levels of modularity and structure, more extensive documentation, and more rigorous implementation of an access control model that meets the needs of defence and national security users. In the 1980s, most of the commercial software systems developed in this period achieved at least the lowest level class of the Orange Book.

In the late 1980s, the governments of Canada and several European countries began working on developing their own security evaluation criteria applicable to software products. The final outcome of their efforts led to the European Information Technology Security Evaluation Criteria, or ITSEC [30]. Similar with the "Orange Book" that was used as a US standard, the ITSEC is intended to be used as an international standard to

assess the level of security assurance offered by computing systems. ITSEC differs from the Orange Book in that security feature requirements and assurance requirements are treated separately.

By the mid-1990s, using either the ITSEC or the Orange Book in the security evaluation of trusted computing systems and related software products had become a critical factor for commercial and government customers. In an effort to harmonize security evaluation procedures and provide a wider market for trusted products, the United States government and the ITSEC supporters agreed on the Common Criteria for Information Technology Security Evaluation [14] that was finalized and received formal international recognition in the late 1990s.

To keep the efforts going in building trusted software systems, Microsoft formed in 1998 an internal Security Task Force to examine the underlying causes of software vulnerabilities. The lessons learned from the task force were accumulated and summarized into a set of recommendations to help software security development. These recommendations form the earliest precursor of the so called security development lifecycle (SDL). As more and more wisdom has been collected over years of security development practises, Microsoft released in early 2004 the first formal version of the SDL that was designated as SDL Version 2.0 in recognition of the fact that many product versions had undergone an earlier (and less formal) SDL process during the era of security pushes. In addition, Microsoft officially committed to apply the formally defined SDL to any future Microsoft products that will need security assurance.

2.3.1.2 The SDL Process

The SDL is a process adopted by Microsoft for the development of software that needs to withstand malicious attacks [29]. It involves series of security-focused activities and corresponding deliverables at each of the phases of typical software development process. Specifically, the SDL activities over a typical software development process can be summarized as follows:

Requirements phase: The security development activities of software requirements phase focus on identifying key security goals and discussing the plan and schedule for security integration in the rest of the development process. Usually, at this phase, a security advisor is assigned to the product team, who is supposed to take the responsibility of communication between the central security team and the product team. Specifically, during the requirement phase, the product team needs to define product overall security goals and features in response to customer demands. The security advisor needs to provide security requirements in compliance with industry security standards such as the Common Criteria. Both the product team and the security advisor need to work on the plan and schedule of security integrations into the development process. The security-focused deliverables of these activities mainly consist of documents describing the security integration plan and corresponding risk analysis.

Design Phase: The SDL activities at this phase mainly consist of defining security design guidelines and conducting attack surface analysis and threat modelling.

Defining security design guidelines should be the first security-focused activity during the design phase. The guidelines typically help in specifying the overall structure of the software from security perspective, and deciding the security design approach that will be used by architects and the specific security techniques that will be adopted to implement required security mechanism and services. Both the security advisor and the software architect team should devote some time to laying down these guidelines before any design task takes place. After laying down the guidelines, the next set of actions consists of analyzing the system's attack surfaces.

The primary goal of attack surface analysis is to identify and reduce as much as possible the number of system entries that are susceptible to software attacks. The attack surface analysis can be conducted through several iterations as the system design is being revised.

In addition of attack surface analysis, Microsoft SDL also requires threat modeling activities during the design stage. Briefly, threat modeling involves assessing and documenting system security risks. Generally, a security advisor identifies the assets that the software must manage and the interfaces by which those assets can be accessed, and

then identifies threats that can do harm to each asset and the estimated likelihood of harm being done. The security advisor also should identify appropriate countermeasures against the threat. The results of threat modeling must be reviewed by the architecture team and help them to enhance system security features if necessary.

Implementation Phase: The SDL process requires at the software implementation stage taking appropriate actions to ensure the correctness of the software code and mitigate high-priority threats. Specifically, the SDL activities that apply in the implementation phase may include the following:

1) Applying coding and testing standards to remove flaws that lead to security vulnerabilities and maximize the likelihood of detecting errors that may lead to software vulnerabilities.
2) Applying static-analysis code scanning tools to discover bad coding patterns that may result in known security vulnerabilities.
3) Conducting manual code reviews to examine source code and detect and remove potential security vulnerabilities.

Verification Phase: At this stage, the software is functionally complete and ready for beta testing. During this phase, SDL process requires concentrating verification efforts on security code reviews beyond those completed in the implementation phase. The purpose of these efforts is to ensure that the final software product meets the customer requirements and allow deeper review of any legacy code that has been brought forward from prior software versions.

Release Phase: Prior to releasing the software product, SDL requires the so-called Final Security Review (FSR) activity to take place. The goal of the FSR is to give the organization's top management an overall picture of the security standing of the software product and the likelihood that it will be able to withstand attack after it has been released to customers. FSR usually involves a review of the software's ability to withstand widely known and newly reported vulnerabilities affecting similar software. Penetration testing is sometimes required to supplement manual security vulnerability reviews.

Support and Servicing Phase: In this phase, software product teams must prepare to respond to newly discovered vulnerabilities from customers. Accordingly, SDL requires a security response process to be defined and employed at this stage. This response process should take care of evaluating reported security vulnerabilities, releasing security advisories and updating software system if necessary.

2.3.1.3 The Future of SDL

It can be foreseen that new ways to attack software will be constantly discovered and security researchers will continue to seek new techniques to address software vulnerabilities that are not addressed by current security techniques. Organizations that wish to build more secure software will have to continue their efforts by finding new ways to make software more resistant to attack and by developing tools and techniques that respond to new classes of attacks when they are discovered. As a result, the SDL process will keep evolving and incorporating new features to respond to the continuing challenge of software security. For instance, the Microsoft SDL standard has continuously been updated every six months since 2004 when the first formal SDL version was released. The SDL Version 2.1 went into effect in January 2005, and Version 2.2 became effective in July 2005. SDL Version 3.0, a major revision that incorporated privacy requirements for Microsoft products, went into effect in January 2006.

The measurement framework proposed in our work can be integrated in the design phase of the SDL process. Our framework can be used both to define suitable architecture security metrics and to apply these metrics to software design artefacts.

2.3.2 Software Architecture Analysis

Security architecture design represents a critical part of the SDL process because most of the serious security issues arise at the design stage. Generally, security investigation at the software architecture level is a difficult task. Security analysts usually need to have both common understanding of high-level software design issues and application-specific

security challenges [37]. Over decades, architectural-level security analysis has relied solely on human expertise and been conducted mainly using *ad-hoc* techniques. Recently, several methodologies have been proposed in the literature for security analysis of software designs. In the remainder of this section, we introduce and discuss some representative instances of these methodologies.

2.3.2.1 Software Architecture Analysis Method (SAAM)

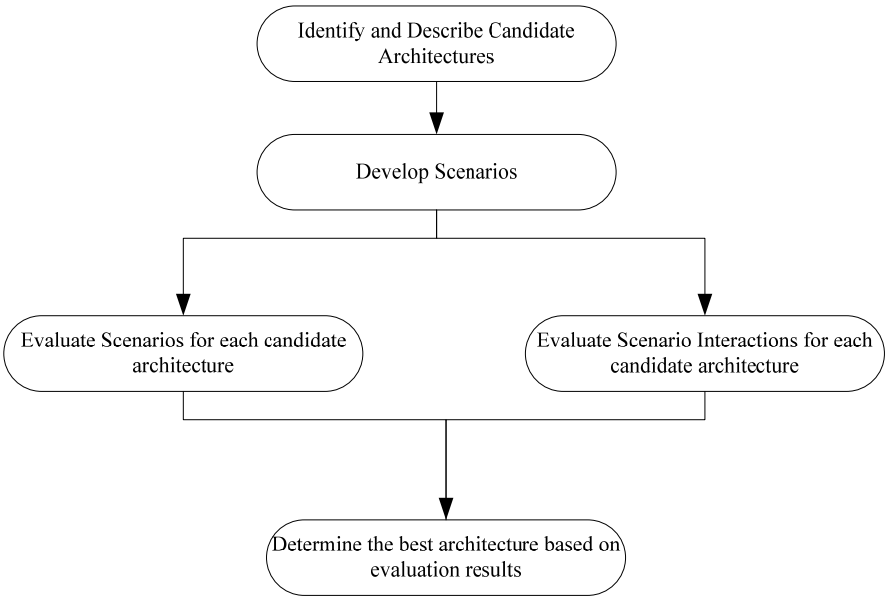

Figure 2.3. SAAM Activities

Kazman et. al proposed in 1996 the Software Architecture Analysis Method (SAAM). SAAM is a scenario-based methodology to evaluate quality factors of software architectures [37]. Specifically, they proposed to define a variety of use scenarios in the application domain and evaluate candidate architectures against each scenario based on human expertise. In this way, instead of using a single indicator to assess an architecture, competative software architectures can be compared on a per-scenario basis. When candidate architectures outscore each other for different scenarios (which is usually the case), the adoption of a particular architecture will depend on the most critical scenarios in the application domain.

Figure 2.3 illustrates the five steps involved in the SAAM activities. We briefly describe each of these steps as follows:

Identify and describe candidate architectures: At this stage, candidate architecture styles are determined and corresponding interpretations are provided. SAAM doesn't require specific forms of architecture artefacts, but requires the architecture description to clearly specify system components and their relationships.

Develop scenarios: Scenarios represent system activities relevant to different system roles. In this step, architecture analysts should capture as much as possible scenarios supported by the system.

Evaluate scenarios for each of the candidate architectures: Scenario evaluation involves two phases. In the first phase, direct and indirect scenarios are identified. SAAM considers a scenario to be an indirect scenario if the execution of the scenario requires changes to the architecture already defined. Otherwise, a scenario is considered to be a direct scenario. The second phase involves listing the changes for each of the indirect scenario and estimating the cost of performing the changes. The final outcome of this step is a summary table that contains all the direct/indirect scenarios and the estimated change cost for indirect scenarios.

Evaluate scenario interactions for each of the candidate architectures: Two indirect scenarios are considered to be interacting with each other if the two scenarios necessitate changes to the architecture on the same components. SAAM requires scenarios interactions to be evaluated with respect to the degree of sharing between scenarios since this can be used to indicate the defect level of the final products.

Determine the best architecture based on evaluation results: Based on the evaluation results from previous steps, the architecture analysts can decide which version of the architecture is more suitable for their purpose. The selection is based on the evaluation results of the most critical scenarios in the application context.

The proposed methodology can be applied to evaluate different kinds of software qualities including security, as long as the security-related scenario profiles are properly constructed. As such, our framework can be used in conjunction with this methodology.

2.3.2.2 Reliability Risk Analysis

Risk is a function defined on the occurrence frequency of an undesired event and the potential severity of the event consequences. There are several types of risk such as availability risk, acceptance risk, performance risk, and cost risk etc. In software engineering, risk assessment helps identifying potentially troublesome software components that require careful development and allocation of more testing effort. Motivated by the need to assess risks at the early stage of software development, Goseva-Popstojanova *et al.* [23] proposed a methodology for reliability risk analysis at the architecture level. The methodology uses dynamic complexity and coupling metrics derived from UML design specification to generate reliability-based risk factors for various design elements including software components, connectors, software scenarios, and the overall software architecture.

Specifically, their analysis is based on design statistics derived from the Unified Modeling Language (UML) design artefacts. First, for each component and connector in the software architecture, they calculate corresponding complexity and coupling metrics and assess dynamic heuristic risk factors based on hazard analysis. Then, they construct a Markov model to obtain scenario risk factors. The identified scenario risk factors serve as the basis to estimate the risk factors for corresponding use cases and the overall system.
In the proposed methodology, design metrics are used as the basis to estimate the risk factors. The resulting risk factors help software architects and analysts to identify the risky aspects of the software system during the design stage, which in consequence can guide later the allocation of development resources. Even though reliability risk factors, by their nature, may not be strong indicators for software security, the proposed methodology may potentially be extended to software security analysis in the sense that security, just like reliability, is also related to the inner structure of software applications.

In particular, the measurement framework proposed in our work can be applied for architectural level security analysis along the same lines.

2.3.2.3 Approaches based on Formal Methods

Over decades of research, formal methods have been used in software requirement engineering, specifications, and design verifications. Formal methods are software development methods that can be used to perform rigorous mathematical specifications and verifications of software systems. However, due to their esoteric nature, formal methods have been facing huge barrier for their adoption by the industry [66]. In practice, formal methods are often used in fields where the benefits of having rigorous proofs or the danger of having undetected errors are worth the underlying efforts and cost, such as security-critical or safety-critical systems. Several methodologies using formal methods to analyze software security properties at the design stage have been proposed [36], [74], [16], [11], [79]. We discuss some of them in the rest of this section.

Jürjens proposed a security specification and validation framework using UML [36]. The framework allows security-related information to be expressed explicitly using UML modeling elements and evaluated using an underlying formal model. Specifically, the framework contains a UML extension, named UMLsec, and formal semantics of a simplified segment of UML. UMLsec complements the standard UML semantics and notation with an additional profile that contains a set of security-related stereotypes, tags and constraints. Software architects and designers can, therefore, import the UMLsec profile using standard UML extension mechanisms and use them to model software artefacts according to specific security concerns. The augmented UML diagrams can then be transformed into formal semantics and verified against the expected security requirements using a formal reasoning system. The goal of the proposed framework is to provide a rigorous verification system by making the formal aspects transparent to users. Accordingly, two factors are critical to the success of the framework, namely whether the formal reasoning process can be fully automated and whether the verification results returned by the formal reasoning system are informative enough to be understood by developers who do not have formal method background. At the current state of the framework, these two aspects still remain unaddressed.

Using Petri net, Xu *et al.* [74] proposed a threat-driven approach to verify software design specifications with respect to anticipated security threats. The proposed approach consists of three main steps. Firstly, software behaviour, security threats and threat mitigations are modeled using the Petri net semantics and notations. Secondly, the properties and consistency between behaviour models and threat models are verified by taking advantages of the Petri net verification mechanisms to be certain that threats are modeled correctly. Thirdly, threat mitigations, behaviour model and threat model are joined together in a combined model, then checked by investigating whether threats are absent from the mitigated behaviour models. Although the proposed framework may help in ensuring formally the immunity of a software design from some anticipated security threats, the formal aspects of the approach are not hidden from the front-end users. This means that expertise in formal methods is required, which is not the case for most architects and developers.

Deng *et al.* [16] presented a framework, named Software Architecture Model (SAM), which can be used to check rigorously application-specific security properties in software design artefacts [70]. The methodology involves five key steps. Firstly, the software architecture is described using SAM. Secondly, system-wide security properties are expressed using security constraint patterns, a generic expression form introduced by them to specify formally system security policies. Thirdly, global security constraints are decomposed according to individual SAM components. Fourthly, consistency between the system-wide security constraints and component-level constraints is checked. Fifth, using temporal logic, verification proceeds by determining whether the system design satisfies the desired security constraints [11].

Lafrance and Mullins introduced, in 2003, a formal approach to validate security protocols against DOS robustness [79]. Their approach consists of verifying whether the designed protocol satisfies an information flow property named *"impassivity"* based on the formal specifications of the protocol's principles and the intruder's attacks. Specifically, the property of *Impassivity* represents an admissible-interference-based security property which helps detect the design flaw related with improper provoked costly actions, such as resource allocation. The satisfaction of the *Impassivity* guarantees that the protocol provides protection against a DOS attack. In their work, they specify the

the protocol's principles and intruder's attacks using the Security Protocols Process Algebra (SPPA).

As mentioned above, approaches based on formal methods face huge challenges for practical adoption due to their esoteric nature. This does not mean that formal methods are not used in industrial products, but rather that industry vendors do not always follow these models in the purely formal and mathematical way that the models were designed for. In this context, security metrics represent an interesting alternative because they are easy to understand and apply, and may be used in a systematic software development process with limited human expertise. Furthermore there is a well-established tradition of using metrics for quality assessment in the software industry.

Chapter 3

Security Attributes and Properties

In this chapter, we define an axiomatic framework for theoretical validation of internal software security metrics. Specifically, based on popular security design principles, we identify a number of security measurement concepts and properties that can be used to derive or validate theoretically related security metrics, then we illustrate the feasibility of our framework by conducting case studies using attack surface measurement system and the privilege graph .paradigm which have been studied in previous research.

3.1 Software Security Concepts

In this section, we summarize and discuss popular security design principles. We use these principles as basis to identify internal security related attributes and define some intuitive hypotheses for empirical study of corresponding measurement properties.

3.1.1 Security Design Principles

A great deal of wisdom regarding secure system development has been gathered in the past years. This has led to the definition of several security design principles, which are currently used for the design and implementation of secure systems [60]. Table 3.1 gives a list of some of the most popular of these principles.

Our goal is to derive from these principles a set of security related attributes and their corresponding properties to guide the definition of software security metrics. The main characteristics of these principles are simplicity, separation, and restriction.

Simplicity is essential in secure systems engineering for obvious reasons: complex

mechanisms are difficult to build, maintain, and use, and, thereby, tend to increase security risks. Simplicity is highlighted by the principles of "economy of mechanism" and "psychological acceptability".

Separation is mainly inspired by the need for sharing in computer systems. Sharing in software systems is usually the source of many security breaches. Even though "sharing" decreases the security in software systems, it is still necessary in most software systems since it usually eases the implementations of software functional requirements. Therefore, a careful design of sharing mechanisms is essential. Separation is highlighted by the "least common mechanism" principle. Restriction is based on the rationale that no one deserves unlimited trusts; people can always misuse the privileges granted to them. Hence, these privileges must be limited to the strict minimum required to fulfill system functionality.

Restriction is highlighted by principles such as "least privilege", "separation of privilege", and "complete mediation". Besides simplicity, separation and restriction, some security design principles also emphasize other aspects such as security composition and the relation between security and fault-tolerance. For instance, the "weakest link" principle refers specifically to security composition. According to this principle, the security of a collection of elements is at best equal to that of the least secure element.

Table 3.1. Security Design Principles

Principles	Definitions
Least privilege	A subject should be granted only the minimum number of privileges that it needs in order to accomplish its job.
Fail-Safe Defaults	The default access to an object is none.
Economy of Mechanism	Security mechanisms of systems should be kept as simple as possible.
Complete Mediation	All accesses to objects must be checked beforehand.
Open Design	Security of a mechanism must neither depend on the secrecy of design nor the ignorance of others.
Separation of Privilege	Permission must not be granted based on a single condition.
Least Common Mechanism	Mechanisms used to protect resources must not be shared.
Psychological Acceptability	The introduction of a security mechanism should not make the system more complex than it is without it.
Weakest link	A system is only as secure as its weakest element.

3.1.2 Software Security Attributes

As indicated earlier, there are two kinds of quality attributes: external and internal. External attributes refer to software qualities whose presence or absence may be detected by stakeholders such as users or customers. Internal attributes correspond to hidden software qualities meaningful only to software professionals (e.g., developers, specifiers, testers) who have access to software work products such as code, specification, or documentation. External attributes can be achieved by judiciously applying techniques internal to software construction ensuring internal qualities. In the rest of this section, we summarize external software security attributes, which are more familiar to users and customers and then, based on the security design principles, we propose a limited number of internal software attributes, which influence directly or indirectly software security qualities. Although the list is by no means exhaustive, it represents a basis for the development of an initial software security metrics program.

External Software Security Attributes: Software security is a complex and multifaceted property, which can be appropriately captured only through many different quality attributes. The notion of software security encompasses both traditional security attributes and classical dependability attributes [58]. Software security involves multiple attributes, such as authentication, authorization, audit, confidentiality, integrity and so on. Some of these attributes, for instance, authentication, authorization, and audit, can be specified as system functional requirements, thus they can be verified and tested as normal system functionalities. Others like confidentiality and integrity correspond to non-functional requirements in engineering secure software. Non-functional requirements are difficult to capture and analyze, and it is quite common that software systems are being developed without serious handling of non-functional security requirements. Security-related dependability attributes include reliability, availability, performability, and safety. Reliability is the probability that a system delivers a specified function during a given time period. Availability metrics the fraction of time during which a system can deliver its intended service in a given time period or in steady state. Safety refers to the probability that the system will operate normally or abnormally in a given time without

causing significant damages. Performability gives metrics of system performance under attacks or failures.

Although these attributes address primarily dependability issues, they also influence at diverse level overall system security. For instance, major sources of security exploits are software bugs that could be fixed by delivering reliable software. Security attacks such as denial of service have a direct impact on system availability. Many instances of cyber-attacks could be considered as threats to safety. For instance, a penetration attack leading to patients' records modification represents a direct threat to their lives. In addition of these attributes, two external security attributes, which are gaining in interest, are survivability and attackability. Survivability refers to the ability of a system to perform its intended mission under attack or failure, in a timely fashion [55]. Attackability is a concept proposed by Howard and colleagues to measure the extent that a software system could be the target of successful attacks [28]. They define attackability as the short word for attack opportunity. More specifically, attackability is the likelihood that a successful attack will happen.

Internal Software Security Attributes: Many security-related internal software attributes can be derived from the security design principles presented earlier. According to these principles a secure system can be studied by assessing three main characteristics: simplicity, separation and restriction. A software system can be viewed or designed as a collection of services working in concert to deliver its business functions. So, it is possible to estimate the level of security of a software application by assessing the degree of simplicity, separation and restriction characteristics of the underlying services designs. In order to better convey our understanding of these characteristics, we consider four internal attributes that are related to software services as follows:

- *Service Complexity:* captures the level of complexity of a software service.
- *Service Coupling:* captures the level of coupling between software services.
- *Service Excess Privilege:* captures the amount of excessive privileges that a software service may grant compared to its required privileges.

- *Service Mechanisms Strength:* capture the combined strength of the security mechanisms protecting a software service.

Several other internal security attributes could be defined, although we limit the focus of this paper to only the above four attributes. We derive these security measurement concepts based on the security design principles. *Service Complexity* is derived from the principles stressing simplicity such as *economy of mechanisms*. *Service Coupling* comes from principles stressing separation such as *least common mechanism*. *Excess Privilege* comes from principles stressing restriction such as *least privileges. Mechanisms Strength* is derived from security mechanism related principles such as *economy of mechanism, open design* and *least common mechanism*.

Actually the internal attributes suggested in this section are not new. Complexity and coupling have been proposed and widely used as internal software attributes, but with a focus on traditional qualities such as reliability and maintainability, not security. Excess privilege was introduced by T. Smith [62] to study privileges at the system level. Mechanism strength has been studied under various names. For instance, Eloff proposed a rating method for security mechanisms such as cryptographic primitives or password mechanisms that can be considered as a way of evaluating the strength of the mechanisms [17]. L. Smith reported on pilot activities to investigate the practicality of developing metrics for determining the strength of cryptographic algorithms [63]. Based on a small sample of algorithms, the pilot defined five algorithm strengths: *US-unconditionally secure, CS-computationally secure, CCS-conditionally computationally secure, W-weak, VW-very weak*. The results of the pilot demonstrated a set of possible metrics for measuring cryptographic strength based on key length, attack time, steps, and rounds. Even though those works do not target specifically software applications, the techniques used can be adapted to compute the strength of the security mechanisms involved in these applications. In this work, we associate mechanism strength with any kind of software service, either security providers or non-security providers. We assume that mechanism strength for individual security mechanisms are specified by vendors or computed using specific techniques such as the ones mentioned in [63]. The mechanism strength for

regular software services can be computed by combining the strength of the individual mechanisms involved.

3.1.3 Hypothesis for Empirical Studies

As mentioned earlier, to improve the software product we need to be able to affect its internal features. Needless to say, internal metrics are easy to collect but hard to interpret while external metrics are easy to interpret but hard to collect. Estimation models based for instance on regression analysis or Bayesian probability allow the mapping of hard to interpret internal measurement data into easily interpretable external measurement data [76]. We can draw from the previous analysis some hypotheses that can be used to analyze empirically the relationship between the internal and external security measurement concepts. More specifically, we define four hypotheses based on intuitive understanding of the security design principles as follows:

- ***Hypothesis 1:*** Security generally decreases as Service Complexity increases.
- ***Hypothesis 2:*** Security generally decreases as Service Coupling increases.
- ***Hypothesis 3:*** Security generally decreases as Service Excess Privilege increases.
- ***Hypothesis 4:*** Security generally increases as Service Mechanism Strength increases

These hypotheses may help in developing the prediction models. The dependent and independent variables may correspond to external and internal attributes respectively.

3.2 Generic Software Model

Typically, a software system involves multiple services. Either legitimate or illegitimate users usually may take advantages of deficiencies and vulnerabilities between these services to carry security attacks. Hence, design and implementation of software services is an essential aspect of software security management. In this section, we present a generic software model that is built on the concept of software service. Our model is an

extension of the basic service-oriented model proposed by Millen for system survivability measurement [55]. In the rest of the section, we use the proposed model as basis to introduce security measurement properties for the internal security attributes suggested earlier.

3.2.1 Basic Model

The primitive building block of a system consists of a collection of components *C* and a set of relationships among the components $R \subseteq C \times C$. This is equivalent to the basic model of Briand *et al.* [8], who use the term elements instead of components. Unlike Briand *et al.,* who use a modular software model, we use in this work a service-oriented software model, which fits well with security analysis purposes. In [55], Millen defines a system as "a set of components configured to provide a set of user services". A configuration consists of a collection of components connected in a specific way, each providing specific services. These services are referred to as the *supporting services* for the corresponding configuration. In Millen's model, a system *S* is defined as a set of service configurations with a partition $\overline{S}$ on *S* representing the set of services involved. Specifically, a service is defined by a set of alternative configurations, where each configuration defines a set of supporting services. The partitioning in configurations defines a hierarchical structure between services. In this hierarchy, terminal services, which are isolated from one another, are considered atomic services. Given a configuration $s \in S$, $\bar{s} \in \overline{S}$ and $\underline{s} \subseteq \overline{S}$ denote the service containing *s* and the support service set for *s,* respectively. An atomic service is a service, which has no supporting services. Formally, $\forall s \in S$, $\bar{s}$ is an atomic service if and only if $\underline{s} = \varnothing$. Services and configurations have the following properties:

(1) $\bar{s} \notin \underline{s}$

(2) $(\overline{s_1} = \overline{s_2}) \wedge (\underline{s_1} \subseteq \underline{s_2}) \Rightarrow s_1 = s_2$

Property (1) stands for the irreflexivity of the support relation, meaning that a service does not support itself. Property (2) stands for the irredundancy of the configurations associated with the same service, meaning that different configurations of the same service cannot have exactly the same support sets. However, different configurations of two distinct services may have the same support set. As a consequence of the irreflexivity property, an atomic service $\bar{s}$ has only one configuration: $\bar{s} = \{s\}$. This is expressed formally by the following proposition.

Proposition 1: $\forall s \in S\ \bar{s}$ *is an atomic service* $\Rightarrow \bar{s} = \{s\}$

Proof :

Since $\forall s \in S\ \bar{s}$ *is atomic* $\Leftrightarrow \underline{s} = \varnothing$, *we simply need to show* $\forall s \in S\ \underline{s} = \varnothing \Rightarrow \bar{s} = \{s\}$.

Assume $\exists s \in S\ \underline{s} = \varnothing \wedge \bar{s} \neq \{s\}$,

Then $\exists s' \in S\ (s \neq s') \wedge (\{s, s'\} \subseteq \bar{s}) \wedge (\underline{s} = \varnothing) \Rightarrow \exists s' \in S\ (s \neq s') \wedge (\bar{s} = \bar{s'}) \wedge (\underline{s} \subseteq \underline{s'})$

That is in contradiction with irredundancy property (2).

Therefore, $\forall s \in S\ \underline{s} = \varnothing \Rightarrow \bar{s} = \{s\}$. □

The relationships between two configurations correspond to the relationships between their respective components. Given a service configuration *s*, let *μ(s)* denote the set of components involved in *s*. The set of relationships between two configurations s_1 and s_2 is denoted and defined as: $s_1 \propto s_2 = R \cap (\mu(s_1) \times \mu(s_2))$. The relationships between two services correspond to the relationships between their respective configurations. We define and denote the set of relationships between two services $\bar{s_1}$ and $\bar{s_2}$ as: $\bar{s_1} \propto \bar{s_2} = \bigcup_{t_1 \in \bar{s_1}, t_2 \in \bar{s_2}} (t_1 \propto t_2)$.

3.2.2 Extended Model

The basic model presented above focuses primarily on the measurement of survivability; in order to capture a wider range of software security attributes, we need to extend it. We extend the basic model by characterizing a software system with security concerns as a tuple <*S*, $\bar{S}_{Requestor}$, *Γ*, *℘*, *priv*, *pol*>. Following Millen's model, *S* denotes a set of service

configurations, with a partition $\overline{S}$ on S. Γ denotes the set of all security mechanisms associated with the software system. A security mechanism can be, for instance, an encryption program, a password-checking program, or a collection of related protection programs etc. In our framework, we view security mechanisms as special kinds of services that focus on protecting other services; accordingly, $\Gamma \subseteq \overline{S}$. Given a software service $\bar{s} \in \overline{S}$ and a security mechanism $r \in \Gamma$, we use $r \rhd \bar{s}$ to denote that the security mechanism r protects $\bar{s}$. Furthermore, security mechanisms can protect a service either in parallel or in sequence. As an example, a software system may use a combination of authentication mechanisms for the verification of identities, such as password and eye-scan. Successful authentication may require either passing at least one of the systems or all of them in sequence. In order to capture this aspect, we introduce two new operators $\oplus : \Gamma \times \Gamma \to \Gamma$ and $\otimes : \Gamma \times \Gamma \to \Gamma$. We use $r_1 \otimes r_2 \rhd \bar{s}$ to denote that r_1 and r_2 work in parallel to protect service $\bar{s}$ and $r_1 \oplus r_2 \rhd \bar{s}$ to denote that r_1 and r_2 work in sequence to protect service $\bar{s}$. We naturally extend notations $\oplus$ and $\otimes$ to sets of security mechanisms: given $m \subset \Gamma$, $\oplus m$ denotes a set of mechanisms working in sequence, while $\otimes m$ denotes a set of mechanisms working in parallel.

$\wp$ denote the set of all privileges associated with the software system. The notion of privileges underlies the notion of rights that a service requestor can have for the operations and resources associated with the service. A service requestor can be a user, an end-user application or another service; we use $\overline{S}_{\mathrm{Re}questor}$ to denote the set of service requestors involved in the system *S*. Privileges may be associated either with a service requestor or with system resources. Privileges associated with a requestor are sometimes referred to in the literature as privilege attributes. There is a variety of privilege attributes including access identity, roles, groups, security clearances, and capabilities. Privileges associated with resources are referred to as control attributes. Examples of control attributes include access control lists and labels. Due to the diversity of the notion of privilege, we approach privilege generically as a primitive undefined concept like in [61]. A privilege in this work may refer to any of the schemes indicated above. A service requestor must have specific privileges in order to be able to access the resources

involved in a given service. We define a privilege function as a function that maps a tuple service requestor-service with the set of privileges (actually) involved. A privilege function is denoted by $priv : \overline{S}_{Requestor} \times \overline{S} \rightarrow 2^{\wp}$, where $2^{\wp}$ is the power set of $\wp$. The privileges associated with a service can be derived from the operations and resources involved. Analogously, we define a security policy as a function that maps a tuple service requestor-service with a set of (allowed) privileges; a security policy is denoted: $pol : \overline{S}_{Requestor} \times \overline{S} \rightarrow 2^{\wp}$. The difference between a security policy and a privilege function is that the former is supplied with the security requirements, and as such its enforcement is mandatory, whereas the latter is derived from the structure of the software product, as such it is a matter of (design) choice.

3.2.3 Example

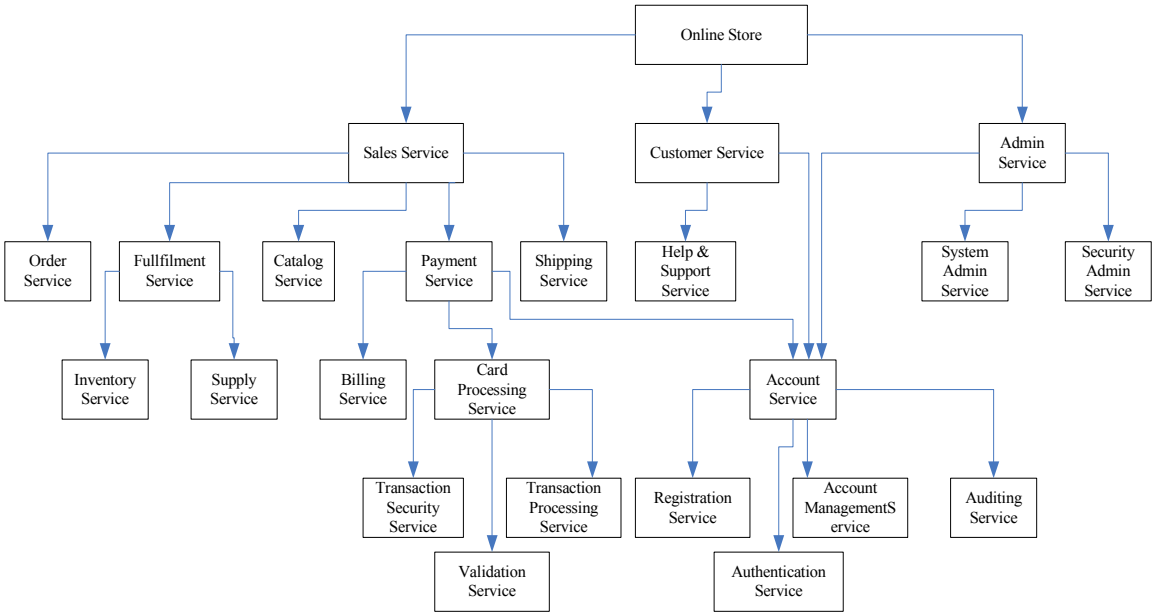

Figure 3.1. Example: Service-oriented architecture for an Online Retail Store. An arrow pointing from service X to service Y indicates that Y is a supporting service for one of the configurations of X.

As an example, Figure 3.1 depicts the service hierarchy for an online retail store. This architecture can be refined further, but as suggested by Millen, the depth of the hierarchy

is a design choice [55]. It is up to the designer to decide the level of granularity of service specification.

In Figure 3.1, arrows denote the support relationships between services. For instance, the root service represented by the online store is supported by three services: sales service, customer service, and admin service. An interesting service in this hierarchy is the account service, which supports three different services: payment service, customer service, and admin service. As indicated earlier, each service consists of one or many different configurations each supported by specific services. For instance, the account service may be defined in terms of two configurations: new user configuration and existing user configuration. A new user needs to register by creating an account. After registration, he may start using the other functions offered by account service without going through authentication; so the support set for this configuration is *{registration service, account management service, auditing service*}. An existing user, in contrast, doesn't need to register, but has to be authenticated. Hence, the support set for existing user configuration is *{authentication service, account management service, auditing service}*. Examples of security mechanisms in this architecture include the (payment) transaction security service, the authentication service, and the auditing service. Services may be implemented as custom components or using COTS technology. For instance, the (payment) transaction security service can be implemented using *secure electronic transactions (SET)* protocol, while the authentication service may be based on a combination of *transport layer security (TLS)* and *LDAP* protocols, which in both cases are COTS technologies.

3.3 Security Measurement Concepts and Properties

In this section, we formally define measurement properties associated with the internal attributes identified previously. These properties will serve as basis for theoretical validation of derived metrics. It is important to stress that the proposed properties represent necessary but not sufficient conditions of validity for related metrics. We are by no means referring to security validity, which is hard to establish, considering the complexity of the notion itself. Establishing completeness of metrics validity is a difficult

task as discussed in [72]. It is also important to emphasize that validity refers here to the validity of the metrics definition with respect to the concept they intended to measure.

3.3.1 Properties of Service Complexity

In [52], Melton *et al.* makes a distinction between psychological complexity and structural complexity. Psychological complexity is based both on system characteristics and human factors. Structural complexity refers to the complexity arising from the software system irrespective of any underlying cognitive considerations. Both forms of complexity affect software security. For instance, the principle of psychological acceptability is related to psychological complexity issues, while the principle of economy of mechanisms is primarily related to structural complexity. However, we limit the scope of this work only to structural complexity; the reader is referred to [52] for detailed definition of this concept. In this work, complexity is defined from the perspective of the services involved in software systems. As indicated in the previous generic software model, a software service may involve various configurations; as a consequence the complexity of a service derives naturally from the combined complexity of its various configurations. Formally, given a software system <*S*, $\overline{S}_{Requestor}$, *Γ*, *℘*, *priv*, *pol*>, the *complexity* of a software service $\bar{s} \in \overline{S}$ is defined by a function denoted by *Complexity*($\bar{s}$). We extend the same notation to service configurations, by defining the complexity of a configuration $s \in S$ as a function *Complexity(s)*. Generally, given a software system <*S*, $\overline{S}_{Requestor}$, *Γ*, *℘*, *priv*, *pol*>, we expect these functions to satisfy the following properties:

1) Axiom AC1: $\forall \bar{s} \in \overline{S}\ Complexity(\bar{s}) \geq 0$

2) Axiom AC2: $\forall \bar{s} \in \overline{S}\ \forall t \in \bar{s}\ Complexity(\bar{s}) \geq Complexity(t)$

3) Axiom AC3: $\forall s \in S\ \forall \bar{t} \in \underline{s}\ Complexity(\bar{t}) \leq Complexity(s)$

4) Axiom AC4: $\forall s \in S\ \underline{s} = \varnothing \Rightarrow Complexity(\bar{s}) = Complexity(s)$

5) Axiom AC5: $\forall \bar{s}, \bar{t} \in \overline{S}\ \bar{s} \subseteq \bar{t} \Rightarrow Complexity(\bar{s}) \leq Complexity(\bar{t})$

We expect the complexity of a service to be non-negative (AC1), and to be no less than any of its configuration complexity (AC2). We expect the *complexity* of a service configuration to be no less than the complexity of any of its supporting services (AC3). An atomic service has a single configuration; we expect the complexity of an atomic service to be equal to the complexity of the corresponding configuration (AC4). If the configuration set of a service is a subset of the configuration set of another service, we expect that the complexity of the former service is no more than the complexity of the latter service (AC5). As a consequence of the complexity axioms, we can make some interesting deductions. For instance, we can deduce that the complexity of a service configuration is nonnegative. Also, we can deduce that the complexity of a service should be no less than the complexity of any of its supporting services. The following theorems express these properties:

6) Theorem TC1: $\forall s \in S \;\; Complexity(s) \geq 0$

Proof :

$$\left.\begin{array}{ll} \forall s \in S \;\; \underline{s} = \varnothing \Rightarrow Complexity(s) = Complexity(\bar{s}) \geq 0 & (\text{AC1,AC4}) \\ \forall s \in S \;\; \underline{s} \neq \varnothing \Rightarrow \forall \bar{t} \in \underline{s} \;\; Complexity(s) \geq Complexity(\bar{t}) \geq 0 & (AC1, AC3) \end{array}\right\} \Rightarrow \forall s \in S \;\; Complexity(s) \geq 0. \;\square$$

7) Theorem TC2: $\forall \bar{s} \in \bar{S} \;\; \forall t \in \bar{s} \;\; \forall \bar{h} \in \underline{t} \;\; Complexity(\bar{s}) \geq Complexity(\bar{h})$

Proof :

$$\left.\begin{array}{ll} \forall \bar{s} \in \bar{S} \;\; \forall t \in \bar{s} \;\; Complexity(\bar{s}) \geq Complexity(t) & (AC2) \\ \forall t \in \bar{s} \;\; \forall \bar{h} \in \underline{t} \;\; Complexity(t) \geq Complexity(\bar{h}) & (AC3) \end{array}\right\} \Rightarrow \forall \bar{s} \in \bar{S} \;\; \forall t \in \bar{s} \;\; \forall \bar{h} \in \underline{t} \;\; Complexity(\bar{s}) \geq Complexity(\bar{h}). \;\square$$

3.3.2 Properties of Service Coupling

The concept of service coupling is used in this work to capture the amount of relationships between services. The notion of relationships between services or configurations has been defined earlier. Formally, given a software system <*S,* $\bar{S}_{Requestor}$, *Γ, ℘, priv, pol*>, the *coupling* between two software services $\bar{s_1}, \bar{s_2} \in \bar{S}$ is defined as a function denoted by *Coupling(* $\bar{s_1}, \bar{s_2}$ *)* and the coupling between two configurations

$s_1, s_2 \in S$ is denoted by *Coupling*(s_1, s_2). The coupling function should satisfy axioms AD1-AD8 listed as follows:

1) Axiom AD1: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ Coupling(\overline{s_1}, \overline{s_2}) \geq 0$

2) Axiom AD2: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \overline{s_1} \propto \overline{s_2} = \varnothing \Rightarrow Coupling(\overline{s_1}, \overline{s_2}) = 0$

3) Axiom AD3: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \forall t_1 \in \overline{s_1}\ \forall t_2 \in \overline{s_2}\ Coupling(\overline{s_1}, \overline{s_2}) \geq Coupling(t_1, t_2)$

4) Axiom AD4: $\forall t_1, t_2 \in S\ \forall \overline{s}_i \in \underline{t_1}\ \forall \overline{s}_j \in \underline{t_2}\ Coupling(t_1, t_2) \geq Coupling(\overline{s_i}, \overline{s_j})$

5) Axiom AD5: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) \leq \sum_{\overline{s}_i \in \underline{s_1}} \sum_{\overline{s}_j \in \underline{s_2}} Coupling(\overline{s_i}, \overline{s_j})$

6) Axiom AD6: $\forall \overline{s_1}, \overline{s_2}, \overline{s} \in \overline{S}\ \overline{s_1} \subseteq \overline{s_2} \Rightarrow Coupling(\overline{s_1}, \overline{s}) \leq Coupling(\overline{s_2}, \overline{s})$

7) Axiom AD7: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ Coupling(\overline{s_1}, \overline{s_2}) = Coupling(\overline{s_2}, \overline{s_1})$

8) Axiom AD8: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) = Coupling(s_2, s_1)$

In other words, we expect *service coupling* to be nonnegative (AD1), and to be zero when there is no relationship between services (AD2). Since services may consist of various configurations, we expect the coupling between two services to be no less than the coupling between any pair of corresponding configurations (AD3). Since the coupling between a pair of configurations is composed of the couplings of the corresponding supporting services, we expect that the coupling between a pair of configurations to be no less than the coupling between any pair of corresponding supporting services (AD4), and no more than the sum of the couplings between all their supporting services pairs (AD5). If the configuration set of a service is a subset of the configuration set of another service, we expect the coupling between the former service and a given service to be no more than the coupling between the same service and the latter service (AD6). Service coupling is symmetric (AD7); so is configuration coupling (AD8). A direct consequence of the service coupling axioms is that the coupling between any pair of configurations should be non-negative. Another consequence of these axioms is that the coupling between a pair of service configurations is zero if all the couplings between their pairs of supporting

services are zero. These properties are expressed by theorems TD1 and TD2 defined as follows:

9) Theorem TD1: $\forall s_1, s_2 \in S \; Coupling(s_1, s_2) \geq 0$

$$\text{Proof}: \left.\begin{array}{ll} \forall t_1, t_2 \in S \; \forall \overline{s}_i \in \underline{t_1} \; \forall \overline{s}_j \in \underline{t_2} \; Coupling(t_1, t_2) \geq Coupling(\overline{s_i}, \overline{s_j}) & (AD4) \\ \forall \overline{s_i}, \overline{s_j} \in \overline{S} \; Coupling(\overline{s_i}, \overline{s_j}) \geq 0 & (AD1) \end{array}\right\} \Rightarrow \forall t_1, t_2 \in S \; Coupling(t_1, t_2) \geq 0. \; \square$$

10) Theorem TD2: $\forall s_1, s_2 \in S \; \forall \overline{t_1} \in \underline{s_1} \; \forall \overline{t_2} \in \underline{s_2} \; \overline{t_1} \propto \overline{t_2} = \varnothing \Rightarrow Coupling(s_1, s_2) = 0$

$$\text{Proof}: \left.\begin{array}{ll} \forall s_1, s_2 \in S \; \forall \overline{t_1} \in \underline{s_1}, \overline{t_2} \in \underline{s_2} \; \overline{t_1} \propto \overline{t_2} = \varnothing & \\ \quad \Rightarrow \forall \overline{t_1} \in \underline{s_1}, \overline{t_2} \in \underline{s_2} \; Coupling(\overline{t_1}, \overline{t_2}) = 0 & (AD2) \\ Coupling(s_1, s_2) \leq \sum_{t_1 \in \underline{s_1}} \sum_{t_2 \in \underline{s_2}} Coupling(\overline{t_1}, \overline{t_2}) & (AD5) \end{array}\right\} \Rightarrow Coupling(s_1, s_2) = 0. \; \square$$

3.3.3 Properties of Service Excess Privilege

Users are assigned some privileges according to the security policy underlying the software system. According to the *least privilege* principle, a user should be granted only the minimum number of privileges needed in order to accomplish a job. However, in reality, the implementation and configuration of software systems usually grant unnecessary privileges to actors for various reasons mainly related to careless design or over simplification. The concept of *Excess Privilege* originally introduced by Terry A. Smith [62] refers to the amount of unnecessary privileges assigned given a specific user domain and task. As mentioned previously, privileges are related to services. From the perspective of software products, the actual privileges granted to users can be derived from the interactions between service requesters and services. Comparing the actual privileges set granted with the minimum set of privileges needed derives *Excess Privilege*. Formally, given a software system <*S*, $\overline{S}_{Requestor}$, Γ, $\wp$, *priv, pol*>, we define the *Excess Privilege* of a software service $\overline{s} \in \overline{S}$, as a function *EP*($\overline{s}$), and the *Excess Privilege* of a

service configuration $s \in S$, as a function *EP(s).* We expect these functions to satisfy axioms AP1-AP5, defined as follows.

1) Axiom AP1: $\forall \bar{s} \in \bar{S}\ EP(\bar{s}) \geq 0$

2) Axiom AP2: $\forall s \in S\ \forall \bar{t} \in \underline{s}\ EP(s) \geq EP(\bar{t})$

3) Axiom AP3: $\forall s \in S\ \underline{s} \neq \varnothing\ \Rightarrow EP(s) \leq \sum_{\bar{t}_i \in \underline{s}} EP(\bar{t}_i)$

4) Axiom AP4: $\forall s \in S\ EP(\bar{s}) \geq EP(s)$

5) Axiom AP5: $\forall s \in S\ \underline{s} = \varnothing \Rightarrow EP(\bar{s}) = EP(s)$

More specifically, we expect the *Excess Privilege* of a software service to be nonnegative (AP1). We expect the *Excess Privilege* of a configuration to be no less than the excess privileges of any of its supporting services (AP2), and no more than the sum of the excess privileges of all of its supporting services (AP3) due to the fact that the same excess privileges may be granted by different supporting services. Since a software service consists of several possible configurations, we expect its excess privilege to be no less than the excess privileges of any of its configurations (AP4). We also expect the excess privilege of an atomic service to be equal to that of corresponding (unique) configuration (AP5). As a consequence of these axioms, theorems TP1 and TP2 defined in the following, state respectively that the *Excess Privilege* of a service configuration should be non-negative, and that the *Excess privilege* of a service should be no less than that of any of its supporting services.

6) Theorem TP1: $\forall s \in S\ EP(s) \geq 0$

$\text{Pr}oof$:

$$\left.\begin{array}{ll} \forall s \in S\ \underline{s} = \varnothing\ EP(s) = EP(\bar{s}) \geq 0 & (AP1, AP5) \\ \forall s \in S\ \underline{s} \neq \varnothing\ \forall \bar{t} \in \underline{s}\ EP(s) \geq EP(\bar{t}) & (AP1, AP2) \end{array}\right\} \Rightarrow \forall s \in S, EP(s) \geq 0. \square$$

7) Theorem TP2: $\forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ \forall \bar{h} \in \underline{t} \Rightarrow EP(\bar{s}) \geq EP(\bar{h})$

$$\text{Proof}:$$
$$\left.\begin{array}{ll} \forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ EP(\bar{s}) \geq EP(t) & (AP4) \\ \forall \bar{h} \in \underline{t}\ EP(t) \geq EP(\bar{h}) & (AP2) \end{array}\right\} \Rightarrow \forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ \forall \bar{h} \in \underline{t}\ EP(\bar{s}) \geq EP(\bar{h}). \square$$

3.3.4 Properties of Service Mechanism Strength

As stated earlier, security mechanisms are considered as special kinds of services that protect other services. In software applications, a service may be restricted by various security mechanisms and some services may deserve more restriction than others. Also, different configurations of a software service may be protected by different security mechanisms. In our framework, the concept of *Mechanism Strength* is used to capture the strength of the security mechanisms involved in the protection of a particular software service. Formally, given a software system $<S, \bar{S}_{Requestor}, \Gamma, \wp, priv, pol>$, we define the *Mechanism Strength* of a software service $\bar{s} \in \bar{S}$, as a function *MStrength*$(\bar{s})$, and the *Mechanism Strength* of a service configuration $s \in S$, as a function *MStrength*(s). We expect these functions to satisfy axioms AS1-AS7, defined as follows.

1) Axiom AS1: $\forall \bar{s} \in \bar{S}\ MStrength(\bar{s}) \geq 0$

2) Axiom AS2: $\forall s \in S\ \underline{s} = \varnothing \Rightarrow MStrength(\bar{s}) = MStrength(s)$

3) Axiom AS3: $\forall s \in S\ [\bar{s} \in (\bar{S} - \Gamma) \wedge \underline{s} = \varnothing] \Rightarrow MStrength(\bar{s}) = 0$

4) Axiom AS4: $\forall s \in S\ \underline{s} \neq \varnothing \Rightarrow [(\forall \bar{t} \in \underline{s}\ MStrength(\bar{t}) = 0) \Rightarrow MStrength(s) = 0]$

5) Axiom AS5: $\forall \bar{s} \in \bar{S}\ (\forall t \in \bar{s}\ MStrength(t) = 0) \Rightarrow MStrength(\bar{s}) = 0$

6) Axiom AS6: $\forall \bar{s} \in \bar{S}, m \subset \Gamma\ \otimes m \triangleright \bar{s} \Rightarrow MStrength(\bar{s}) \leq \underset{r \in m}{Min}\{MStrength(r)\}$

7) Axiom AS7: $\forall \bar{s} \in \bar{S}, m \subset \Gamma\ \oplus m \triangleright \bar{s} \Rightarrow MStrength(\bar{s}) \geq \underset{r \in m}{Max}\{MStrength(r)\}$

We expect the mechanism strength of a software service to be nonnegative (AS1). We expect the mechanism strength of an atomic service to be equal to that of its (unique) configuration (AS2). We also expect that the mechanism strength of an atomic service which is not part of Γ to be always zero (AS3). Given a service configuration, when none

of the supporting services are protected by some security mechanism, we naturally expect the mechanism strength of such configuration to be zero (AS4). Similarly, when none of the configurations of a service is protected by some security mechanisms, we expect the mechanism strength of such service to be zero (AS5). Based on the weakest link principle, which states that a system is not more secure than its weakest component, we expect the mechanism strength of a service protected by a collection of security mechanisms in parallel to be no more than the minimum strength of the protecting mechanisms (AS6). When a service is protected by a collection of mechanisms in sequence, its mechanism strength is expected to be no less than the maximum strength of the protecting mechanisms (AS7). A natural and obvious consequence of the mechanism strength axioms is that when there are no security mechanisms protecting the software system, the mechanism strength of each involved service is zero. This is expressed by theorem TS1:

8) Theorem TS1: $\Gamma = \varnothing \Rightarrow \forall \bar{s} \in \bar{S}\ MStrength(\bar{s}) = 0$

Pr*oof* :

Let the system S have a n-layer *hierarchical structure*
whose root node (i.e., layer 0) i*s the top service* represented by *system S*,
and bottom nodes (i.e., layer n-1) *represent the atomic configurations*.
Let $\bar{S}_i \subseteq \bar{S}$ $(i = 2m,\ m = 0,\ 1,\ 2,...)$
represent the set of services in the ith *layer of the system S*.
Let $S_j \subseteq S$ $(j = 2m+1,\ m = 0,\ 1,\ 2,\ ...)$
represent the set of configurations in the jth *layer of the system S*.

(1) $\Gamma = \varnothing$

$\Rightarrow (\forall s \in S\ \ \underline{s} = \varnothing \Rightarrow (MStrength(s) = MStrength(\bar{s})) \wedge (MStrength(\bar{s}) = 0))$ $(AS2, AS3)$

$\Rightarrow (\forall s \in S, \underline{s} = \varnothing \Rightarrow MStrength(s) = 0)$

$\Rightarrow (\forall s \in S_{n-1}\ MStrength(s) = 0)$

(2) $\forall$k (k ≥ 3), a*ssume that* $\Gamma{=}\varnothing \Rightarrow \forall s \in S_k\ MStrength(s) = 0$,
then we can deduct the following :

$(\Gamma{=}\varnothing \Rightarrow \forall \bar{s} \in \bar{S}_{k-1}\ \ MStrength(\bar{s}) = 0\ )$ $(AS5)$

$\Rightarrow (\Gamma{=}\varnothing \Rightarrow \forall s \in S_{k-2}\ \ MStrength(s) = 0)$ $(AS4)$

3.4 Case Studies

In this section, we illustrate the properties introduced above by presenting case studies based on existing measurement frameworks. The first study, based on the attack surface metrics proposed in [51], [28], allows the derivation and evaluation of sample coupling, complexity and excess privileges metrics. The second study, based on the privilege graph paradigm [15], allows the derivation and evaluation of sample mechanism strength metrics.

3.4.1 Attack Surface Metrics System

In this Section, we give an overview of the attack surface framework and related metrics, and discuss how the framework can be expressed using our software model. Then, from the basic metrics introduced in [51], we derive a collection of metrics that match three of our internal attributes, namely service coupling, service complexity, and excess privilege.

Notion of Attack Surface: The system's *attack surface* consists of the combination of the system actions externally visible to the users and the resources accessed or modified by these actions. The more actions or resources are available to the users, the more exposed the system is to successful attacks, and so the more insecure it is. Underlying the notion of attack surface is the notion of an attack class. An attack class categorizes resources based on specific properties, for instance, the group of services running as root or the group of open sockets. The rationale behind attack classes is that some categories of resources offer more attack opportunities than other; for instance, services running as root are more exposed than those running as non-root. Attack classes provide common ground for comparing different versions of the same system or different systems exhibiting the same sets of attack classes. In [51], [28], the attack surface metrics system is defined as a state machine $<S,I,A,T>$, where S is the set of states, I is the set of initial states ($I \subseteq S$), A is the set of actions, defined in terms of *pre* and *post* conditions, and T is the transition relation. A state is defined as a triple $(e,s,am) \in S$, where e represents the environment consisting of a mapping of names to typed resources, s is called the store and

corresponds to a mapping of typed resources to their typed values, and *am* is an access matrix that is defined as a triple *Principal×Resource×Rights* based on Lampson's access matrix model [62]. It is assumed that the set of resources *Resource* is partitioned into disjoint typed sets. The set of actions A is partitioned into four sets A_S, A_T, A_A, and A_U corresponding to the action set of the system, the action set of the Threat, the action set of the Administrator, and the action set of the User, respectively. The attack surface metrics system as presented, can easily be described using our service-oriented software model $<S, \overline{S}_{Requestor}, \Gamma, \wp, priv, pol>$. More specifically, software services encapsulate resources (set *Resource*), and actions (set *Action*). Service requestors correspond to subjects (set *Principal*). Service configurations can be derived from action execution sequences. Γ and *priv* can be derived from the *pre* and *post* conditions of the actions involved in the attack surface metrics system. The policy function *pol* and the set of privileges $\wp$ can be derived from the access matrices of the attack surface system.

Attack Surface Metrics: In [51], the attack surface is defined as a tuple $\left(A_S, \bigcup_{a \in A_S} Res(a) \right)$, where A_S is the system action set and *Res(a)* is the set of resources associated with action *a*. The attack surface contribution is measured as the weighted sum of the number of instances of each attack class. The weights are computed by defining a payoff function *payoff: Attack_Class→[0,1]*. Formally, the attack surface is defined as:

$$attack\_surf = \sum_{i=1}^{n} count(S_i) \times payoff(S_i) \qquad \text{(Metric 1)}$$

Where S_i is an attack class, $count(S_i)$ returns the number of instances of the attack class S_i, and n is the number of top attack classes (based on the payoffs). In [51], it is suggested that not all attack classes need to be involved in the metric of an attack surface.

Derived Metrics: The attack surface metric presented in the previous section is based on internal software characteristics (e.g., actions, resources), and as such can be used to

easily derive internal metrics based on our framework. Specifically, in this section, we derive three metrics, each associated with one of the internal attributes defined earlier. The first derived metric is related to service complexity, and obtained by restricting the Metric 1 to software services and configurations. Specifically, given a set of top attack classes S_{Attack}, the attack surface of a software service $\bar{s}$ (resp. a configuration s) can be defined as follows:

$$attack\_surf_1(\bar{s}) = \sum_{a \in S_{Attack}} count(a \bullet \bar{s}) \times payoff(a)$$

$$attack\_surf_1(s) = \begin{cases} \sum_{a \in S_{Attack}} count(\bigcup_{t \in \underline{s}} (a \bullet \bar{t})) \times payoff(a) & (\text{if } \underline{s} \neq \varnothing) \\ attack\_surf_1(\bar{s}) & (\text{if } \underline{s} = \varnothing) \end{cases} \qquad \text{(Metric 2)}$$

Where $a \bullet \bar{s}$ (resp. $a \bullet s$) denotes the set of attack instances of type a related to service $\bar{s}$ (resp. configuration s), and *count(x)* represents the size of set x. The instances of attack classes are recognized from the inner characteristics of a service. Like metric *attack_surf*, metric $attack\_surf_1$ is based on the number of instances of top attack classes. Top attack classes are the most critical from a security perspective. So the higher the number of instances of these classes the more complex the tasks of the security services. So, we can assume that $attack\_surf_1$ is equivalent to a complexity metric.

The second derived metric is related to service coupling. As indicated above the action set of the attack surface metrics framework consists of the actions of the System, the Threat, the Administrator and the User. The Threat represents the adversary who attacks the system with malicious objectives. The User represents the principal who uses the system to fulfill legitimate purposes. Commonly, an adversary attacks a software system by trying to manipulate available system resources such that the software services provided to regular system users are affected. Accordingly, a possible requirement that can be derived from the attack surface framework is that the system resources shared by the Threat and the User should be minimal. Based on this requirement, we can derive an attack surface metric by counting the resources that are available to both the Threat and the User of the system. Specifically, given a service $\bar{s}_t \in \bar{S}$ (resp. a configuration $s_t \in S$)

that is available to a threat, and a user service $\overline{s_u} \in \overline{S}$ (resp. a configuration $s_u \in S$), an attack surface metric can be defined as follows:

$$attack\_surface_2(\overline{s_t}, \overline{s_u}) = cardinality(\bigcup_{t_i \in s_t} resource(t_i) \cap \bigcup_{t_j \in s_u} resource(t_j))$$

$$attack\_surface_2(s_t, s_u) = \qquad \text{(Metric 3)}$$

$$\begin{cases} cardinality(\bigcup_{t_i \in s_t} resource(\overline{t_i}) \cap \bigcup_{t_j \in s_u} resource(\overline{t_j})) & (\text{if } s_t \neq \varnothing \wedge s_u \neq \varnothing) \\ cardinality(\bigcup_{t_i \in s_t} resource(\overline{t_i}) \cap resource(\overline{s_u})) & (\text{if } s_t \neq \varnothing \wedge s_u = \varnothing) \\ cardinality(resource(\overline{s_t}) \cap \bigcup_{t_j \in s_u} resource(\overline{t_j})) & (\text{if } s_t = \varnothing \wedge s_u \neq \varnothing) \\ attack\_surface_2(\overline{s_t}, \overline{s_u}) & (\text{if } s_t = \varnothing \wedge s_u = \varnothing) \end{cases}$$

Where $resource(\overline{s})$ (resp. *resource(t)*) is the set of system resources associated with service $\overline{s}$ (resp. the configuration *t*). Since $attack\_surface_2$ gives a metric of sharing between services, intuitively we can assume that it is equivalent to a coupling metric.

The third metric derived from the attack surface framework is related to excess privileges. Three kinds of privileges are associated with the delivery of a software service to a given requestor: *actual privileges, allowed privileges*, and *least privileges* [62]. The actual privileges represent the collection of privileges inherent in the system design, and as such correspond to the entire range of privileges potentially available to the requestor. The allowed privileges represent the set of privileges legally available, which typically are specified by the system security policy. The least privileges represent the minimum set of privileges required to fulfill a given task [60], [62]. In many software systems, for simplicity, the security policy may grant more privileges to some users than the minimum privileges set needed to deliver the service. Also, due to careless design or inadvertently, the actual privileges associated with the delivery of software services may include illegal privileges that are not allowed by the security policy. As mentioned in [51], there is a natural relation between privileges available to users and the resources available to them through the execution of system actions or services. Intuitively, the more unnecessary privileges are granted by a software system, the more attack surfaces may be exposed. So, from the user viewpoint, reducing system actions and easy-to-attack resources is

equivalent to eliminating unnecessary system privileges, which by definition correspond to excess privileges. In [51], the privileges involved are expressed under the form of $Principal \times Resource \times Right$ and the actual privileges granted by a software system can be identified from the pre, post conditions of the system actions. Based on this consideration, an attack surface metric can be defined by taking for a given service the difference between the least privileges required and the actual privileges granted. Specifically, given a software service $\bar{s}$, we can derive an attack surface metric as follows:

$$attack\_surface_3(\bar{s}) =$$
$$cardinality(\bigcup_{u \in principle(\bar{s})} (\bigcup_{res_i \in resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s})))$$
$$attack\_surface_3(s) = \qquad \text{(Metric 4)}$$
$$\begin{cases} attack\_surface_3(\bar{s}) & (\text{if } \underline{s} = \varnothing) \\ cardinality(\bigcup_{\bar{t} \in \underline{s}} \bigcup_{u \in principle(\bar{t})} \{ \bigcup_{res_i \in resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t}) \}) & (\text{if } \underline{s} \neq \varnothing) \end{cases}$$

Where *principle(*$\bar{s}$*)* is the set of resources associated with service $\bar{s}$, *resource(*$\bar{s}$*)* is the set of resources associated with service $\bar{s}$, $right(u,res_i)$ is the set of rights available to the requester u for resource res_i through $\bar{s}$, and *LP(u,* $\bar{s}$*)* is the least privileges set required for requestor u to execute service $\bar{s}$. Intuitively, $attack\_surface_3$ is equivalent to an excess privilege metric.

3.4.2 Privilege Graph Paradigm

Foundation: It has been established by Dacier and Deswartes that the vulnerabilities of a computing system can be described using a privilege graph [15]. A privilege graph is a directed graph in which nodes represent the privileges owned by a user or a group of users, and edges represent potential privilege transfer. Specifically an arc from node n_1 to node n_2 corresponds to an attack method that can allow the owner of the privileges in n_1 to obtain those in n_2. Privilege graphs can be used to derive attack scenarios describing how intruders misuse available privileges and obtain illegal privileges. In [15], Dacier and

Deswartes use the so-called Intrusion Process State Graph to interpret attack scenarios. The Intrusion Process State Graph can be constructed based on the privilege graph of a software system. Specifically, an Intrusion Process State Graph can be defined as directed graph $q=(N_q, I_q, G_q, T_q)$, where N_q represents a set of states, I_q represents the initial state ($I_q \in N_q$), G_q represents the goal state ($G_q \in N_q$), and T_q represents a transition relation ($T_q \subseteq N_q \times N_q$). In an Intrusion Process State Graph, G_q corresponds to the compromised state for an attacker, and T_q corresponds to the methods of privilege transfer. Each of the intrusion process states is associated with a set of privileges owned by an attacker; we denote the set of privileges of an intrusion process state *n* by *privileges(n)*. Each transition is represented by an estimated success rate of the corresponding attacks.

The MTTF Metrics: In [15], the various attack methods involved in a privilege graph are rated using two variables, namely Time and Effort. The transition rate is associated with the mean effort or the mean time to succeed in corresponding attack. Based on the Intrusion Process State Graph, a Markovian model is used to estimate the mean time or effort to reach the target of the corresponding intrusion process. Specifically in [15], the *Mean Time To Failure* (MTTF) metric is used to characterize the mean time for a potential intruder to reach the target in a specific intrusion process. The MTTF metric is defined as the sum of the mean sojourn time of the states involved in the attack paths. Formally, given an intrusion process state graph $q=(N_q, I_q, G_q, T_q)$, the corresponding MTTF metric is defined as follows:

$$MTTF(i) = t_i + \sum_{k \in out(i)} P_{ik} \times MTTF(k)$$
$$t_i = \sum_{k \in out(i)} 1/\lambda_{ik} \qquad P_{ik} = \lambda_{ik} \times t_i \qquad \text{(Metric 5)}$$

Where, t_i represents the mean sojourn time in state *i* that is defined as the inverse of the sum of state *i*'s output transition rate; *out(i)* denotes the set of output transitions from state *i*; λ_{ik} is the transition rate from state *i* to state *k*; P_{ik} denotes the conditional probability transition from *i* to *k*.

Derived Metrics: Each state of an intrusion process state graph exposes in fact one or more services. The initial state involves potentially a collection of services available to an intruder, while the goal state corresponds to the services targeted in the attack process. The MTTF metric evaluates the difficulty for an intruder to reach his target from a specific state in a given attack scenario. Consequently, we can use the MTTF metric as a metric of the protection strength of a service under specific attack scenario. Since a service may be targeted in various intrusion scenarios, considering the weakest link principle which states that a system is as secure as its weakest protection scheme, the security strength of the service could be represented by the minimum MTTF value. Since a configuration consists of several supporting services, the minimum MTTF value of a configuration could be defined as the sum of the minimum MTTF values of its supporting services. More specifically, given a service $\bar{s} \in \bar{S}$ and the set of intrusion process state graphs $Q_{\bar{s}}$ whose targets represent the service $\bar{s}$, we define a *Minimum MTTF metric* (MinMTTF) for service $\bar{s}$ (resp. configuration *s*) as follows:

$$MinMTTF(\bar{s}) = \underset{q \in Q_{\bar{s}}}{Min}\{MTTF(I_q)\}$$

$$MinMTTF(s) = \begin{cases} \sum_{\bar{t}_i \in \underline{s}} MinMTTF(\bar{t}_i) & (\text{if } \underline{s} \neq \varnothing) \\ MinMTTF(\bar{s}) & (\text{if } \underline{s} = \varnothing) \end{cases} \qquad \text{(Metric 6)}$$

Since *MinMTTF* metric gives a measure of the service protection, intuitively we can assume that it is equivalent to mechanism strength metric.

3.4.3 Properties Verification

The four metrics proposed in the previous sections are based on intuitive understanding of corresponding concepts. We assume that *attack_surf*$_1$, *attack_surf*$_2$, *attack_surf*$_3$, and *MinMTTF* metrics provide measures for the attributes of complexity, coupling, excess privilege and mechanism strength respectively. Now, by checking corresponding measurement properties, we may either increase our confidence in these assumptions or

identify and remedy possible flaws. The verification of a metric may either reveal some flaws in definition, or simply fail. In the former case revising or amending the definition can remedy the metric. In the latter case the metric is considered invalid. In this section, we review and discuss for each metric an example of property highlighting flaws or insufficiencies in the above definitions[1]. For each set of metrics, we first introduce some postulates, which extend the initial definitions as remedies, and then we introduce the proofs of the properties based on these postulates. Without these postulates the verification of corresponding properties would have been difficult or simply unsuccessful.

Metrics attack_surface$_1$: Initial verification of attack_surface$_1$ metrics is successful for all the service complexity properties, except axiom AC2. In order to prove this property, we need to introduce the following postulate:

$$Postulate\ P11:\ \forall a \in S_{attack}\ \forall s \in S\ \forall \bar{t} \in \underline{s}\ [(a \bullet \bar{t}) \subseteq (a \bullet s)] \wedge [(a \bullet s) \subseteq (a \bullet \bar{s})]$$

Postulate P11 states that the set of attack instances related to a service configuration is a subset of the attack instances of corresponding service. Similarly the attack instances of a service configuration include the attack instances of corresponding support services. Based on this postulate the following proof can be given for AC2:

$$\text{Pr}oof:$$

$$Postulate\ P11 \Rightarrow \forall s \in S\ \left[\bigcup_{\bar{t} \in \underline{s}} (a \bullet \bar{t})\right] \subseteq (a \bullet \bar{s})$$

$$\Rightarrow \forall a \in S_{Attack}\ count(\left[\bigcup_{\bar{t} \in \underline{s}} (a \bullet \bar{t})\right]) \leq count(a \bullet \bar{s})$$

$$\Rightarrow \forall a \in S_{Attack}\ count(\left[\bigcup_{\bar{t} \in \underline{s}} (a \bullet \bar{t})\right]) \times payoff(a) \leq count(a \bullet \bar{s}) \times payoff(a)$$

$$\Rightarrow \sum_{a \in S_{attack}} count(\left[\bigcup_{\bar{t} \in \underline{s}} (a \bullet \bar{t})\right]) \times payoff(a) \leq \sum_{a \in S_{attack}} count(a \bullet \bar{s}) \times payoff(a)$$

$$\Rightarrow \forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ attack\_surf_1(t) \leq attack\_surf_1(\bar{s}).\ \square$$

[1] Verification results for the complete set of properties can be found in the Appendix A

Metrics attack_surface$_2$: Initial verification of attack_surface$_2$ metrics is successful for all the service coupling properties, except axioms AD3, AD4 and AD5, which require the following postulates:

$$Postulate\ P21:\ \forall \bar{s} \in \bar{S}\ resource(\bar{s}) = \bigcup_{t \in \bar{s}} resource(t)$$

$$Postulate\ P22:\ \forall s \in S\ resource(s) = \bigcup_{\bar{t} \in \underline{s}} resource(\bar{t})$$

P21 states that the resources of a service correspond to the collection of resources involved in its configurations; P22 states that the resources associated with a configuration corresponds to the union of the resources associated with corresponding support services. We illustrate these concepts by giving the following proof for property AD5:

$Proof$:

1) $Postulate\ P22 \Rightarrow \forall s_1, s_2 \in S\ attack\_surface_2(s_1, s_2) = resource(s_1) \cap resource(s_2)$

2) $Postulate\ P22 \Rightarrow \forall s_1, s_2 \in S\ resource(s_1) \cap resource(s_2) = \left[\bigcup_{\bar{s}_i \in \underline{s_1}} resource(\bar{s}_i)\right] \cap \left[\bigcup_{\bar{s}_j \in \underline{s_2}} resource(\bar{s}_j)\right]$

$$\Rightarrow resource(s_1) \cap resource(s_2) = \bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} \left[resource(\bar{s}_i) \cap resource(\bar{s}_j)\right]$$

$$\Rightarrow cardinality(resource(s_1) \cap resource(s_2)) = cardinality\left(\bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} \left[resource(\bar{s}_i) \cap resource(\bar{s}_j)\right]\right)$$

$$\Rightarrow cardinality(resource(s_1) \cap resource(s_2)) \leq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} cardinality(resource(\bar{s}_i) \cap resource(\bar{s}_j))$$

1) *and* 2) $\Rightarrow \forall s_1, s_2 \in S\ attack\_surface_2(s_1, s_2) \leq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} attack\_surface_2(\bar{s}_i, \bar{s}_j)$. □

Metrics attack_surface$_3$: Initial verification of the excess privileges properties for attack_surface$_3$ metrics is successful except for axiom AP4 that can be satisfied only after introducing the following postulate:

$$Postulate\ P31:\ \forall s \in S\ \forall \bar{t} \in \underline{s}\ \forall u \in \Pr inciple(\bar{t})$$
$$(\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})) \subseteq (\bigcup_{res_i \in Resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s}))$$

P31 states that the excess privileges of a service include the excess privileges of its supporting services. As illustration, we give the following proof for property AP4:

$\Pr oof$:

$$\left.\begin{aligned} &1)\ \forall s \in S\ \underline{s} \neq \varnothing \Rightarrow \bigcup_{\bar{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\bar{t})} (\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})) \\ &\qquad \subseteq \bigcup_{u \in \Pr inciple(\bar{s})} (\bigcup_{res_i \in Resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s}))\ \ (P31) \\ &\Rightarrow cardinality(\bigcup_{\bar{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\bar{t})} \{\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})\}) \\ &\qquad \leq cardinality(\bigcup_{u \in \Pr inciple(\bar{s})} \{\bigcup_{res_i \in Resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s})\}) \\ &\Rightarrow attack\_surface_3(s) \leq attack\_surface_3(\bar{s}) \end{aligned}\right\}$$

$$2)\ \forall s \in S\ \underline{s} = \varnothing \Rightarrow attack\_surface_3(s) = attack\_surface_3(\bar{s}) \qquad (by\ definition)$$

$$1)\ and\ 2) \Rightarrow \forall s \in S\ attack\_surface_3(s) \leq attack\_surface_3(\bar{s}).\ \square$$

Metrics MinMTTF: For these metrics, only three of the mechanism strength properties pose some difficulties during verification, namely axioms AS5, AS6 and AS7. In this regard, we define the following postulate and lemmas:

$$Postulate\ P41: \forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ MinMTTF(\bar{s}) = \underset{t \in \bar{s}}{Min}\{MinMTTF(t)\}$$

$$Lemma\ L41:\ \forall \bar{s} \in \bar{S}\ \forall q \in Q_{\bar{s}}, MTTF(I_q)\ \leq \underset{i \in path(q)}{Min}\{MTTF_i(I_q)\}$$

$$Lemma\ L42:\ \forall \bar{s} \in \bar{S}\ \forall q \in Q_{\bar{s}}\ \forall i_1, i_2 \in path(q)\ \ i_1 \subseteq i_2 \Rightarrow MTTF_{i_1}(I_q)\ \leq MTTF_{i_2}(I_q)$$

Postulate P41 states that the MinMTTF value for a given service corresponds to the minimum MinMTTF value of its different configurations. Lemma L41 states that the MTTF of an intrusion process state graph is always lower than the mean time of its shortest path. Lemma L42 states that the MTTF of a path of an intrusion process state

graph increases as the number of arcs in the path increases. Lemmas L41 and L42 are basic properties of MTTF metric; see [15] for corresponding proofs. Now, based on L41, L42, and P41, we can establish that MinMTTF metrics satisfy axioms AS5, AS6 and AS7. As illustration, we give the following proof for AS6:

$\text{Pr}\,oof$:

Let $i \bullet p$ *denote a subpath of path* p *where* $i \bullet p$ *ends at the node* i *of* p.

$1) \forall \bar{s} \in \bar{S}, m \subset \Gamma \ \otimes m \triangleright \bar{s}$

$\Rightarrow (\forall q \in Q_{\bar{s}} \ \forall r_i, r_j \in m \ r_i \neq r_j \Rightarrow r_i, r_j$ *protect* $\bar{s}$ *in different paths of* q)

$\Rightarrow \forall q \in Q_{\bar{s}} \ \forall i \in path(q) \ \forall r \in m,$ if r *is contained in path* i, $MTTF_{r \bullet i}(I_q) = MTTF_i(I_q)$.

$2)$ *Lemma* $L41$

$\Rightarrow MinMTTF(\bar{s}) = \underset{q \in Q_{\bar{s}}}{Min}\{MTTF(I_q)\} \leq \underset{q \in Q_{\bar{s}}}{Min}\{\underset{i \in path(q)}{Min}\{MTTF_i(I_q)\}\}$

Based on 1) ,2), *we can deduct the following* :

$\forall \bar{s} \in \bar{S}, m \subset \Gamma \ \otimes m \triangleright \bar{s}$

$$\Rightarrow MinMTTF(\bar{s}) \leq \underset{q \in Q_{\bar{s}}}{Min}\{\underset{i \in path(q)}{Min}\{MTTF_{r \bullet i}(I_q)\}\}$$
$$= \underset{r \in m}{Min}\{\underset{q \in Q_{\bar{s}}}{Min}\{MTTF_{r \bullet i}(I_q)\}\}$$
$$= \underset{r \in m}{Min}\{MinMTTF(r)\}. \square$$

3.4.4 Discussion

For better communication among the different stakeholders and for better evaluation processes, it is essential to define unambiguously the various concepts, assumptions, and abstractions underlying software metrics definitions. It is easy to define some new metrics, which intuitively would seem to make sense (in the first place) but when scrutinized closely through theoretical investigation may appear actually to be illogical. In the above case studies, we have been able to uncover insufficiencies in the definitions of sample metrics derived from some existing metrics systems. We follow an iterative process, during which possible limitations are identified in the proposed definitions by checking them against the proposed measurement properties. Two possible outcomes are considered: either it is possible to address the identified weaknesses by modifying or reinforcing the assumptions, in which case the revised definitions undergo new rounds of

verification, or the metrics definitions are considered invalid, and as such are rejected. In all the examples presented above, the initial metrics definitions have to be revised by reinforcing underlying assumptions; otherwise the sample metrics would be considered invalid. These case studies illustrate that the proposed formal framework can be used to evaluate existing software security metrics and assist in the search or design of new ones.

3.5 Summary

Theoretical validation of software security metrics is an important step towards their general acceptance. By using an axiomatic approach, software attributes can be formally defined with a set of ground truth. In this chapter, we have defined and formalized a collection of properties characterizing security-related internal software attributes. The derivation of the attributes and their corresponding properties are based on security design principles widely accepted in the security engineering community. This represents an important contribution in the field of software security metrics, as the field is still in infancy. Our framework provides a ground for identifying and evaluating new and existing security metrics. Like in most previous works on measurement properties [72], our proposed formal framework is sound but not complete. Due to the complexity of the notion of software security, completeness is difficult to establish. Our properties should be interpreted as necessary but not sufficient conditions for validity of related software metrics (in the sense of measurement theory). Note that we are referring here to the validity of the metrics definitions, not to security validity. Also, it is important to note that the internal attributes suggested in this chapter cover only a limited view of software security. However, we plan in future work to extend our framework by investigating more internal attributes and properties in order to further the confidence level in the validation process.

Chapter 4

Security Measurement Abstraction: the USIE Model

In this chapter, we introduce a new security measurement abstraction named *User System Interaction Effect* (USIE) model that can be used systematically to derive security concerns from service-oriented software architecture. The difference between the generic service-oriented model introduced in Chapter 3 and the USIE model to be introduced in this chapter is that the generic service-oriented model abstracts the fundamental elements involved in a service-oriented system, but the USIE model provides a particular representation for the design specifications of a concrete service-oriented architecture. We start by reviewing some background information on service-oriented architecture and discussing underlying security challenges. Then we highlight the modeling objectives of the USIE paradigm, and introduce its semantic model, associated notation, and some illustrative examples.

4.1 Service-Oriented Analysis and Design

In this section, we give an overview of service-oriented architecture, and present as illustration a sample service-oriented application. Then, we present SOA development methodology and discuss related security issues.

4.1.1 Overview

The Service-Oriented Architecture (SOA) as introduced in [18] is an emerging architectural style for building next-generation enterprise applications. Simply put SOA is an architecture in which functionalities of a software system are constructed and delivered as services to either end-user applications or other services. Service-oriented analysis and design (SOAD) combines elements and practices adopted from object-oriented design, component-based design, enterprise software architecting, and business process modeling. It also incorporates distinct mechanisms that facilitate the expression of functional and non-functional requirements for software services. SOAD supports well-established architectural principles such as modularization, separation of concerns and information transparency. In addition, it provides a basis for addressing concerns such as changeability, maintainability, and reusability. As recognition of the importance and benefits of SOAD as an engineering approach for developing enterprise applications, IBM has decided recently to adopt it as a standard development approach, and begun implementing supporting technologies and tools. According to Gartner [3], *"by 2008, more than 60 percent of enterprises will use SOA as a 'guiding principle' when creating mission-critical applications and processes."*

4.1.2 Sample Service-Oriented Application

To illustrate the concepts and models introduced in this chapter, we use as running example a Flower Shop (FS) application, which is a web-based application allowing customers to purchase flowers online. The FS application has been designed and implemented as open source service-oriented software system [2]. The FS application is based on three-layer architecture: user, services, and data. It consists of various services remotely accessible by customers on the Internet. The individual services implemented using PHP are deployed on a web server; confidential information is stored on a backend database server. The FS database consists of 8 tables used to organize information such as user accounts, flower supplies, online sessions, and so on. 38 PHP pages that can be remotely accessed by both online customers and system administrators under proper

privileges implement the various FS services. Figure 4.1 depicts the service hierarchy of the FS application. In this figure, dual circles represent higher level services (referred to as *composite services*) and single circles represent the lowest level services (referred to as atomic services). More details are given on the FS architecture in subsequent sections.

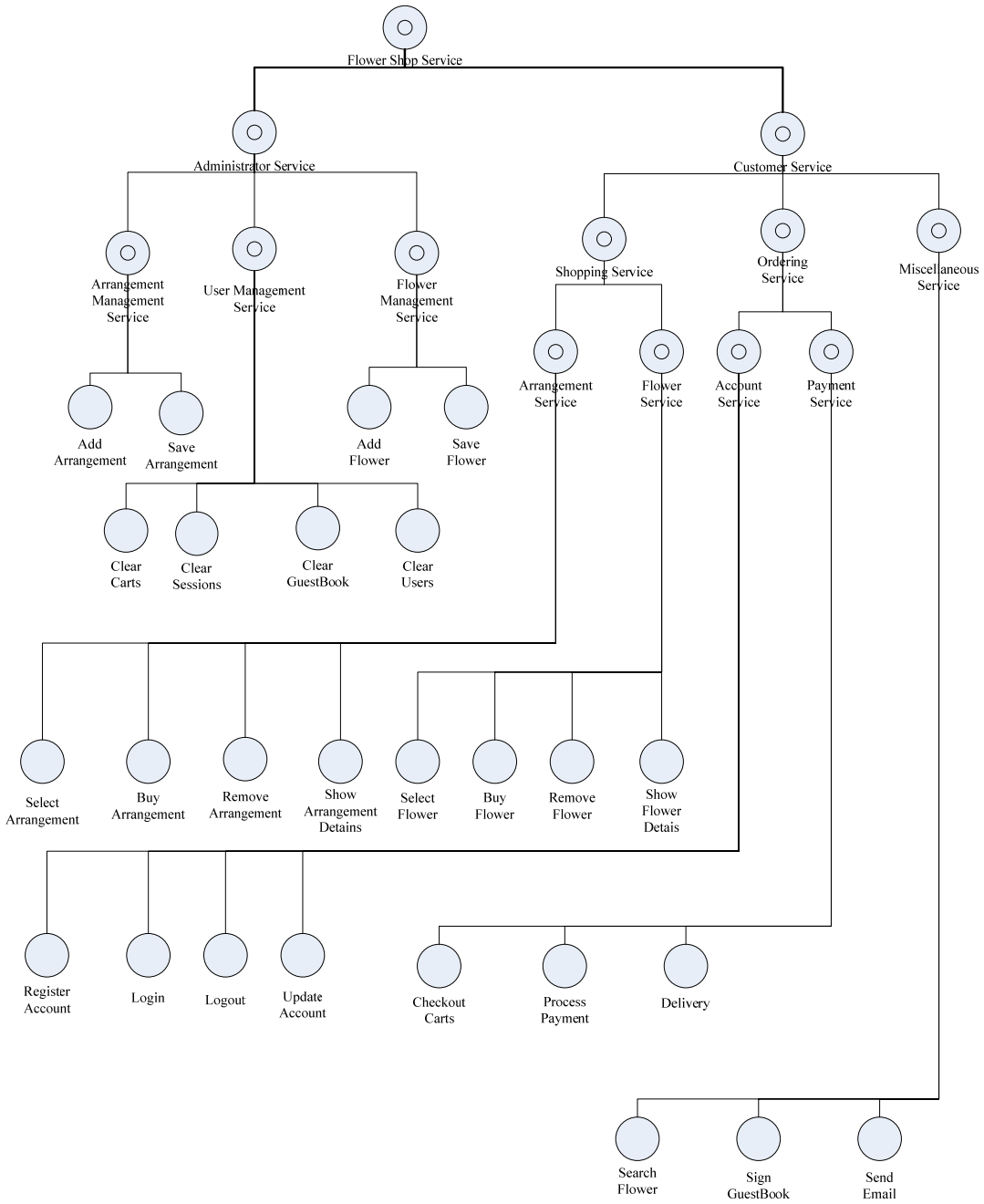

Figure 4.1. Service Hierarchy of FS Application

4.1.3 SOAD Methodology

Conceptually, the SOAD methodology involves three major abstractions: *Business Processes, Services* and *Operations* [75]. *Business Processes* abstraction is used to describe business functions or activities usually identified by domain experts or customers. A business process usually consists of a series of services that are executed in sequence according to a set of business rules. *Services* abstraction is used to describe the collection of services underlying each of the business processes. Typically, a software service under the service-oriented architecture is composed either of other services or by a collection of logically related operations that correspond to logical transactions among system components. *Operation* abstraction is used to describe the set of logical operations involved in the services. Basically, the development of *Business Processes, Services* and *Operations* follow a hierarchical pattern as shown in Figure 4.2, which illustrates partially the architecture of the FS sample application introduced earlier.

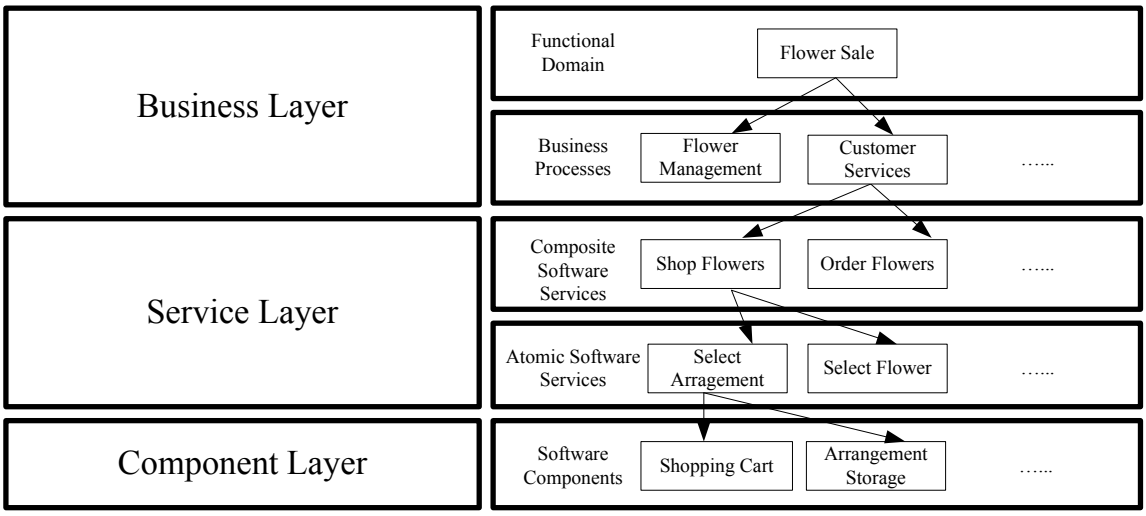

Figure 4.2. The SOAD Hierarchy and Reference Model

So far, a number of techniques and principles have been used to design and implement operations, services, and business processes within SOAD [18], [3], [75]. Traditional software development technologies such as Component Oriented Design (COD) and Object Oriented Design (OOD) are still important and applicable within the SOAD framework. In practice, business processes or functions are normally implemented

or designed as *composite services*, which in turn are composed possibly of other composite services and *atomic services*. An atomic service usually consists of a collection of SOA operations that can be mapped into either a component interface or a specific object-oriented (OO) method with a structured interface. Design and realization of the components with their composition objects are still carried out using traditional COD and OOD techniques.

The UML [6], enhanced with distinct stereotypes and profiles, has become a key notation used to capture SOAD abstractions [3], [34]. Using UML, conceptual artifacts such as business processes and system services can be expressed in concrete and visual models, which are more understandable to different stakeholders. For instance, while specifying SOAD models, the business processes are usually illustrated using UML use case diagrams. A SOA service is usually specified using a UML interaction diagram (i.e., sequence or collaboration diagram). A SOA operation can be expressed using the message semantics underlying the UML interaction diagram. The SOAD specific concerns can be addressed by defining adequate UML stereotypes and profiles.

4.1.4 Security Issues in Service-Oriented Designs

Security assessment of a software product at the design level still requires some expertise and subjective effort. From the standpoint of engineering, subjective assessments are mostly unreliable and inaccurate and should be avoided whenever possible; instead, more systematic evaluation methods should be employed. In this context, developing techniques allowing systematic and automatic software security analysis of SOAD design artifacts is a necessity. However, security analyses of UML models, on which SOA designs are often based, involve some challenges. Specifically, two major challenges arise when using UML service specifications for systematic security analyses. Firstly, the standard UML notation lacks features for capturing satisfactorily security-related events. Based on the standard UML notation, it is hard to derive security concerns of software services such as the privileges granted by the service, the security mechanisms involved and so on. Secondly, expressing instead of reasoning is the primary purpose of UML

diagrams; therefore, it is very difficult (for a computer) to conduct directly security analysis on UML notations.

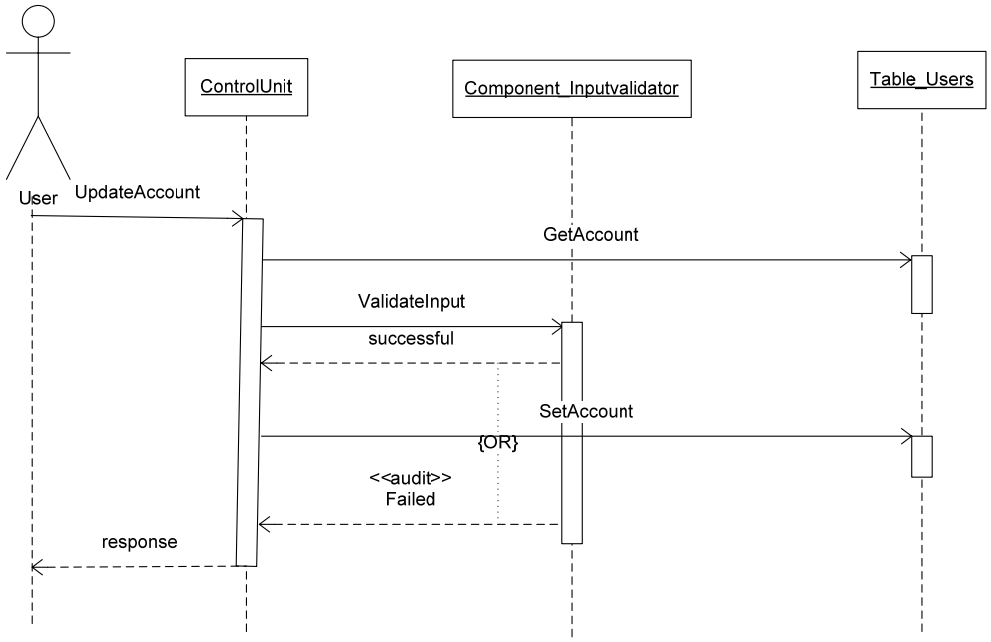

Figure 4.3. Service Specification for *UpdateAccount*

In order to address some of these challenges, various security-related features have been proposed to specify security aspects within software design process [34], [5], [36]. For instance, many security design patterns have been proposed in the literature in order to capture best practices design mechanisms; a summary of currently available patterns can be found in [5]. In the context of SOAD, Johnston, an IBM senior architect, introduced a set of primitive modeling features that allow security intent to be explicitly captured when specifying software services [34]. These consist of dedicated stereotypes and profiles augmenting the standard UML notation. As an example, Figure 4.3 describes FS *UpdateAccount* service, which allows a user to update account information. The service execution is specified using a stereotyped UML sequence diagram. The service execution starts by the submission of *UpdateAccount* request by the service requestor, followed by the validation and processing of the request message. The specification is annotated using security stereotypes suggested by Johnston [34] to capture underlying security intents. For instance, in Figure 4.3, <<*audit*>> indicates that the *ControlUnit*

component will record a "Failed" event; a candidate profile for the <<*audit*>> stereotype is also defined in [34].

Unfortunately, modeling security intent is not enough to address software security concerns. Even though software architects, by applying proper security modeling techniques, can effectively communicate security requirements, the actual design may still convey important security flaws. As an example, the service specification in Figure 4.3 relies on an "Audit" mechanism; a decision has to be made to either design an independent audit component or use the audit functionality provided by the operating system. Such design decisions could affect the overall security strength of the software service. In order to optimize the design of a software system from a security perspective, it is necessary to be able to evaluate the security levels of alternative designs for the same system. This need is also emphasized by the fact that many security attacks could be conducted successfully against service designs such as the one illustrated in Figure 4.3. For instance, based on this design, some of the attack scenarios that may be successful include (among others) *dictionary* and *denial of service (DOS)* attacks. A dictionary attack may be conducted against the *ValidateInput* operation allowing an illegitimate user to steal the credentials of a legitimate one. A DOS attack may simply bring down the server, preventing legitimate requestors from accessing the provided service.

In this context, there is an urgent need beyond modeling of security intent, to be able to isolate and analyze systematically such intent in order to identify and remove potential security design flaws. We address these challenges by proposing in this chapter the USIE model as a paradigm to abstract security intents and events from SOA designs for the purpose of systematic security analysis.

4.2 Goals and Scope of USIE Modeling

As indicated earlier, UML is one of the key notations used for SOA modeling. Also as discussed earlier, although UML is one of the most popular Architecture Description Languages (ADL) it barely supports security artifacts. Security related events could not be described satisfactorily using the basic or regular semantics of UML diagrams. Moreover, the modeling features provided by the UML interaction diagrams make it

rather difficult to conduct directly security analysis. The goal of the USIE model is to address these deficiencies by isolating and capturing security intents and events, and expressing such information in a machine-processable form for automated security analysis of software services designs. More specifically, a USIE model captures firstly the security dependencies underlying the hierarchical structure of software services, and secondly the traces of the communications in user-system interactions as defined by UML interaction diagrams. Such information can be derived automatically from UML models by defining or reusing adequate security profiles. The collected information can also be used to compute and analyze adequate security metrics with the purpose of improving architecture design.

In this work, the USIE paradigm is used as a particular abstraction to capture security concerns from the design of SOAD services. However, its usage is not limited to SOAD models. It can be adapted and used effectively to abstract user system interactions in any kind of software development process such as SAAM [37] and SDL [24].

4.3 USIE Elements

The modeling elements associated with USIE models can be classified broadly into three categories, namely *entities, links* and *branches*. We present, in this section, the semantics and syntax associated with each of these modeling elements.

4.3.1 USIE Entities

Based on the hierarchical structure introduced in chapter 3 for our generic software model, we can identify for a service-oriented system four primary kinds of modeling entities: composite services, configurations, atomic services, and components. A composite service represents a service that is composed of one or more configurations. A configuration is simply represented as a collection of services. An atomic service represents a basic service unit. A component corresponds to a single logical component providing one or more logical operations. As specified by Millen [55], atomic services can have different interpretations according to the modeling concerns.

In order to capture the above concepts, we define in the USIE model six kinds of modeling entities named as follows: *USIE composite service, USIE configuration, USIE atomic service*, *USIE security mechanism*, *USIE resource,* and *USIE null entity*. Each of these modeling entities is represented as a node in a corresponding USIE graph. Figure 4.4 illustrates the graphical notations used to represent them.

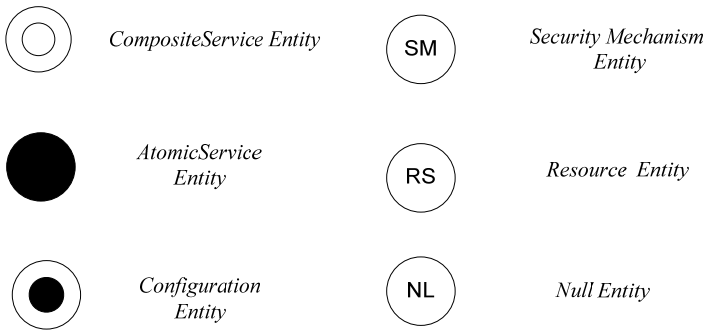

Figure 4.4. Graphical Notations for USIE Entities

To capture service-level concepts underlying service-oriented architecture, we use three kinds of modeling entities, namely USIE atomic service, USIE configuration, and USIE composite service. Specifically, a USIE atomic service entity is defined as a one-time user-system interaction, which corresponds to a single request from a user to the system. For instance, a password-based authentication service may be considered as USIE atomic service in the sense that the service only requires one user-system interaction corresponding to the submission of user's identification and password information to the system. Since a configuration consists of supporting services, a USIE configuration entity therefore can be represented as a collection of user-system interactions. Since a composite service is based on a set of alternative configurations, a USIE composite service entity therefore represents a set of alternative user-system interactions groups, each group corresponding to a specific USIE configuration entity.

At the component level, we use three types of component entities, namely *USIE resource*, *USIE security mechanism*, and *USIE null*. More specifically, we consider a software component to be a USIE security mechanism if it provides or implements some form of protection during a service execution. A component is considered to be a USIE resource if it represents a form of system resources needing some protection. A USIE null

entity component is defined in USIE models to represent a pseudo component for logical purpose. For instance, in Figure 4.3, the service execution is specified using a stereotyped UML sequence diagram. The service execution starts by the submission of *UpdateAccount* request by the service requestor, followed by the validation and processing of the request message. In this service specification, whether or not the *setAccount* operation executes depends on the execution results of the *ValidateInput* operation. In other words, there will be no further operation execution if the *ValidateInput* operation returns an unsuccessful result. This no-operation logic can be expressed by using a dummy operation in USIE model. Specifically the dummy operation can be represented in USIE model by using a USIE operation targeting a USIE null entity. Although, in practice, there are many other categories of software components, the USIE model only considers resource and security mechanisms because these are the types of components involved in a typical security analysis process.

4.3.2 USIE Links

USIE links connect USIE entities in a USIE model. They are expressed using a combination of three different types of modeling elements, namely *USIE composition link*, *USIE operation*, and *USIE service dependency*.

Figure 4.5. Graphical Notation for USIE Composition Link Element

USIE composition link elements are used in a USIE model to connect USIE service entities with configuration entities in order to express the support relationships between services and configurations. Specifically, we define a USIE composition link element as a directed arc connecting USIE service and configuration entities. For instance, if the source of a composition link element is a service entity and the target is a configuration entity, this indicates that the target is a configuration of the source. Likewise, if the source of a composition link element is a configuration entity and the target is a service entity,

this indicates that the target is a supporting service of the source. Figure 4.5 depicts the graphical notation of the USIE composition link element, where the end with a solid circle connects its target and the other end connects with its source.

In the USIE model, links categorized as USIE operation elements correspond to the operations directly constituting USIE atomic services. Specifically, a USIE operation is a request to a USIE component during the execution of a USIE atomic service. A USIE atomic service may consist of one or more USIE operations. When modeling a USIE operation, we are interested primarily in the effect of the underlying operation on components. From a security perspective, the effect of an operation is based on a collection of rights that a service can have on the corresponding component. For illustrative purpose, we specifically consider in this work that the set of rights offered by a software system is a subset of $\{read, write, execute, create, delete\}$. Specifically, we define a USIE operation element as a directed arc with a right attribute. Figure 4.6 illustrates the graphical notation used to represent a USIE operation element. The directed arc connects a USIE atomic service entity, which is the source of the arc, to a USIE component entity, which is the target. The right attribute is defined as a set of rights involved in the execution of the corresponding operation.

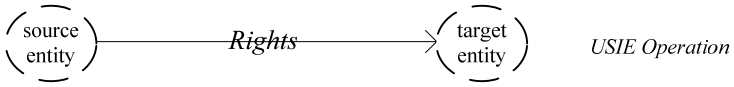

Figure 4.6. Graphical Notation for a USIE Operation Element

In the USIE model, USIE service dependency elements are used to capture the relationships between software services. In practice, some services can be active only after specific services have been activated. As an example, in an on-line banking application, a customer may not be granted any banking service unless he successfully passes the authentication service. We refer to this particular relationship between services as service dependency. More precisely, given two services $\bar{s}_i$, $\bar{s}_j$, service $\bar{s}_i$ is said to be dependent on service $\bar{s}_j$ if a precondition for $\bar{s}_i$ to be active is that $\bar{s}_j$ must be active. Specifically, we define a USIE service dependency element as a directed arc between

USIE service entities as illustrated by Figure 4.7. The notation indicates that the source service depends on the target service.

Figure 4.7. Graphical Notation for a Service Dependency Element

4.3.3 USIE Branches

In addition to USIE entities and link elements, the relationships among the USIE link elements are also the subject of USIE modeling. These relationships are captured using USIE *branches*. There are two categories of link relationships that are of interest in USIE modeling, namely the relationships among USIE operation elements and the relationships among USIE service dependency elements.

When studying the relationships among USIE operations from a security perspective, we are interested particularly in the execution order of these operations during a corresponding service execution. In practice, different kinds of execution order can be considered in software systems. In this work, we consider the following types of execution orders: *Sequential, Concurrent* and *Exclusive*. Specifically, the execution order relationship between two USIE operations is considered to be *Sequential* if one of the operations has to be executed before the execution of the other one starts. In contrast, two operations are in *Concurrent* order if they can be activated concurrently. Two operations are *Exclusive* if they cannot coexist in a single service execution (for instance, in the if-else condition).

In terms of the relationships among USIE service dependencies, we are specifically interested in the logical combinations of the multiple dependencies that a service may have on several other services. Various logical combinations may exist among these dependencies. The simplest and most common forms of logical combinations are *And* and *Or*. For instance, an authentication service may involve password checking and

fingerprint checking, either in sequence or in parallel. As a result, services granted after authentication may become active only after either succeeding all the authentication processes or succeeding only one of them. In the current USIE model, we consider only two basic forms of logical combination of service dependencies, which are *And* and *Or* relationships; future work will investigate other forms of logical relationships.

In general, we define a USIE branch element as a pair *(L, r)*, where *L* represents a set of related USIE link elements and *r* is a label indicating the specific relationship among the set of link elements of *L*; we define $r \in \{Sequential, Concurrent, Exclusive, And, Or\}$. Figure 4.8 illustrates the graphical notation for a *USIE* branch element. The notation consists of a diamond node relating a common source entity to various target entities through specific USIE links. The diamond node is annotated using the value of the label *r*. A solid line links the common source entity to the diamond node. The diamond node in its turn connects to the target entities using the notation corresponding to their specific links (as illustrated above). There is no ordering requirement for the USIE link elements connected by a USIE branch except for a "Sequential" label. A USIE "Sequential" branch requires its USIE link elements be ordered counter clockwise. The leftmost outgoing link from the diamond node will be invoked first, and so on and so forth.

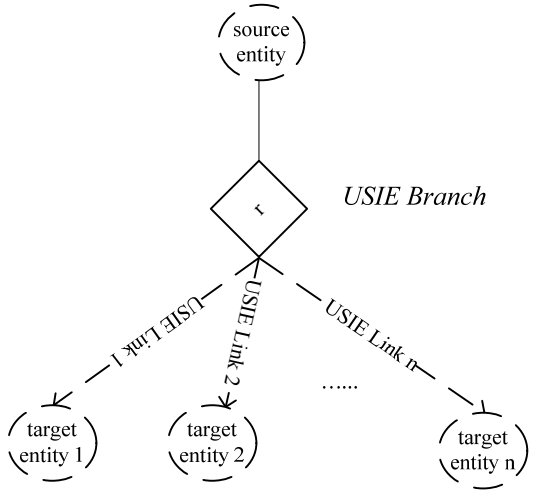

Figure 4.8. Graphical Notation for a USIE Branch Element

4.4 USIE Graphs

A USIE model provides an abstraction of the security features associated with software services. Specifically, a service-oriented software system can be represented by a collection of USIE graphs, each representing a corresponding software service or service configuration. Based on the USIE elements introduced in the previous section, we can define a USIE graph U as a tuple $(N, E, <, L)$, where N represents a set of nodes corresponding to USIE entities, $E \subseteq N \times N$ represents a set of directed edges, $<$ is a partial order relation over E, and L represents a set of USIE branches characterizing the relationships between the edges in E. As explained in the abstract software model introduced earlier, a software system can be represented through a hierarchical service structure, in which composite services are defined in terms of a set of alternative configurations. Configurations are defined in terms of their supporting services, and atomic services are defined in terms of the collection of operations involved. Accordingly, three different kinds of graphs, which capture this hierarchical structure, are defined in the USIE model, namely *atomic service graph, configuration graph*, and *composite service graph*. In practice, the USIE graphs of a service-oriented system can be either constructed directly by software architects or derived systematically from the software architecture description produced using an Architecture Description Language (ADL) notation such as the UML.

4.4.1 USIE Atomic Service Graph

As explained above, a USIE atomic service consists of a collection of USIE operations. The USIE graph corresponding to an atomic service is defined as a tree, in which the root node corresponds to a USIE atomic service entity representing the service itself, and leaf nodes correspond to USIE component entities (e.g., *USIE security mechanism, USIE resource and USIE null entity*). The directed edges in a USIE atomic service graph represent the USIE operations associated with the USIE components. The relationships between related USIE operations are captured using USIE branches. The algorithm presented in Table 4.1 defines explicitly the steps to construct a USIE atomic service

graph given an atomic service $\bar{s}$, and corresponding sets *O* and *L* of USIE operations and branches, respectively.

Table 4.1. Algorithm for USIE Atomic Service Graph Construction

FUNCTION : USIE_AtomicService_Graph_Construction
(An Atomic Service $\bar{s}$, set of USIE Operations O, set of USIE Branches L)

RETURNS a USIE atomic service graph $G(\bar{s})$;

INPUTS : An Atomic Service $\bar{s}$;
set of USIE Operations $O = \{o_1, o_2, \ldots o_n\}$;
set of USIEBranches $L = \{l_1, l_2, \ldots l_m\}$;
INITIALIZATION : $i = 0, G(\bar{s}) = null$;
DO insert an Atomic Service entity $\bar{s}$ into $G(\bar{s})$ as the root node ;
REPEAT : $i \leftarrow i + 1$;
DO insert o_i into $G(\bar{s})$;
UNTIL $i = n$;
RESET $i = 0$;
REPEAT : $i \leftarrow i + 1$;
DO insert l_i into $G(\bar{s})$;
UNTIL $i = m$;
RETURN : $G(\bar{s})$

According to the algorithm of Table 4.1, the construction of a USIE atomic service graph starts by defining a root node corresponding to the atomic service entity. Then the nodes corresponding to the USIE operations involved in the service execution are created in the graph one by one. After that, the graph is annotated by defining wherever necessary suitable USIE branches.

To illustrate USIE model construction for an atomic service, we consider as an example the *UpdateAccount* atomic service illustrated by the sequence diagram of Figure 4.9. Figure 4.9 illustrates the user-system interactions underlying the *UpdateAccount* service by using appropriate stereotypes to highlight the USIE components and operations involved. Specifically, for USIE modeling, we need to identify security mechanisms and resources among the components involved in the user-system interactions, and we need to

specify the right attributes for the operations. As such information is based on design-decisions, it can be provided by the software architects at the design stage of software development. For instance, in the user-system interactions underlying the *UpdateAccount* service, the component named *Component_InputValidator* is stereotyped as a security mechanism using notation <<*SMComponent*>> considering that it carries the implementation of user account information validation process. The *Table_Users* component is annotated as a resource component using stereotype <<*RSComponent*>> since it implements the storage of the sensitive information involved in this service execution. Likewise, the right attributes of the USIE related operations can also be specified with respect to their effects on the targeted USIE components. As shown in Figure 4.9, the *GetAccount* operation is annotated by a stereotype <<*R*>> simply because it is supposed to get some information from its target component. The *ValidateInput* operation has a right stereotype <<*E,R*>> since it causes its target component to execute some logics and return some results. The *SetAccount* operation is meant to modify the state of its target component; therefore it is assigned the right stereotype <<*W*>>.

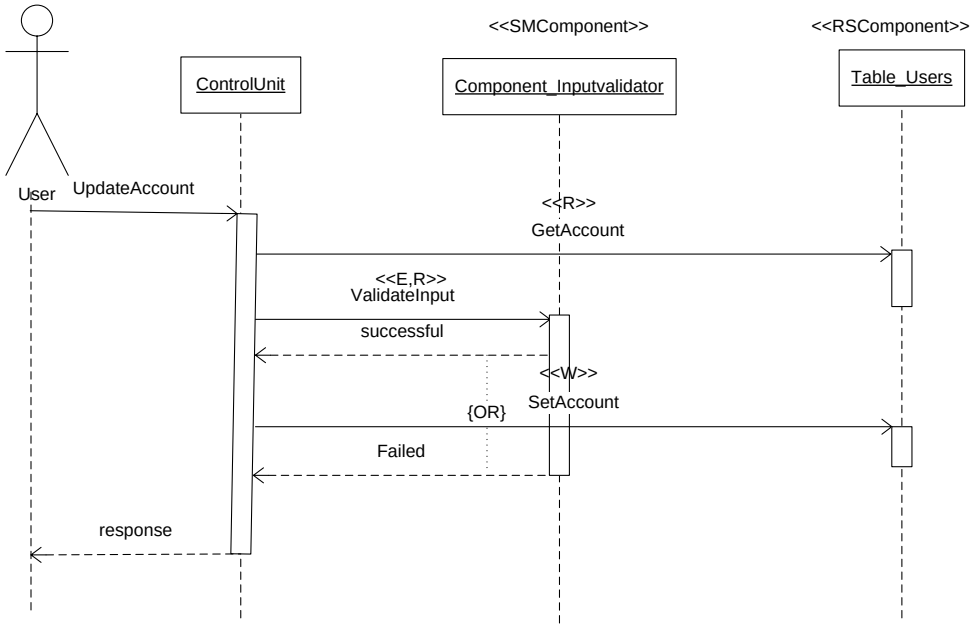

Figure 4.9. Annotated Service Specification for UpdateAccount

Figure 4.10 depicts the USIE model derived from the sequence diagram shown in Figure 4.9 by applying the Algorithm presented in Table 4.1. We use the graphical notations introduced previously. Specifically, the construction of the USIE model starts by creating the root node, which is a USIE atomic service entity named after the corresponding sequence diagram. As illustrated by Figure 4.10, three USIE operations are involved in the atomic service execution, namely *GetAccount, ValidateInput* and *SetAccount*. Each of these operations is represented by a separate USIE operation linking the root node to corresponding target components using the notation provided previously.

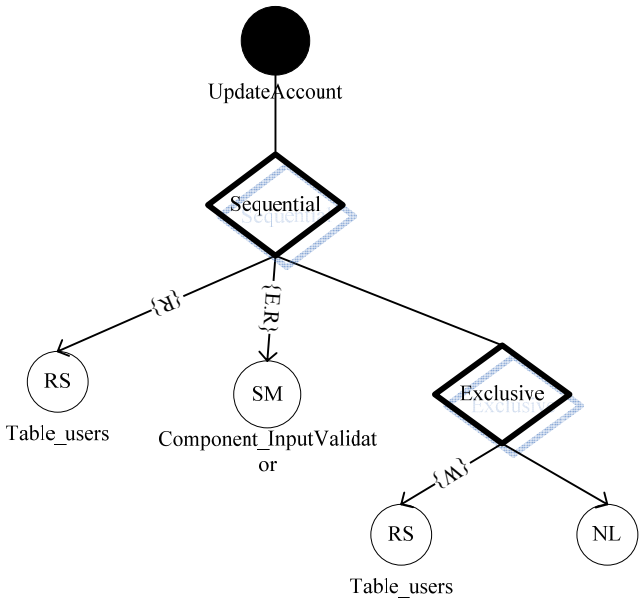

Figure 4.10. USIE Model Derived from the Sequence Diagram of UpdateAccount Service

For instance, in Figure 4.10, the *GetAccount* operation is implicitly represented as a read operation targeting the *Table_users* component, which corresponds to a USIE Resource component. It is important to note that no explicit names are used to represent the operations in the USIE model (in contrast with a typical UML interaction diagram). From the service specification of Figure 4.9, we can notice that the three USIE operations are executed in sequence. As a result the corresponding USIE links can be related using a "Sequential" branch. In particular, an "Exclusive" branch is used to further characterize

the *SetAccount* operation, by associating it with a dummy operation. This captures the fact that the execution of the *SetAccount* operation depends on whether the results of the *ValidateInput* operation is successful or not (as specified by Figure 4.9). Notably, the *ControlUnit* component of Figure 4.9 is not modeled in the USIE model of Figure 4.10 since this component is considered to be neither a security mechanism nor a resource in the service specification.

4.4.2 USIE Configuration Graph

Table 4.2. Algorithm for USIE Configuration Graph Construction

FUNCTION : USIE_Configuration_Graph_Construction(set of Atomic Service $\overline{S}$*,*
set of Service Dependency D,
set of USIE Branch L)
RETURNS a USIE configuration graph $G(\overline{S})$*;*

INPUTS : set of Atomic Service $\overline{S} = \{\bar{s}_1, \bar{s}_2, \ldots \bar{s}_n\}$*;*
set of Service Dependency $D \subseteq \overline{S} \times \overline{S} = \{d_1, d_2, \ldots, d_h\}$*;*
set of USIE Branch $L = \{l_1, l_2, \ldots, l_m\}$*;*
INITIALIZATION : $i = 0, G(\overline{S}) = null$*;*
REPEAT : $i \neg i + 1$*;*
DO insert the atomic service entity of $\bar{s}_i$ *into* $G(\overline{S})$*;*
UNTIL $i = n$*;*
RESET $i = 0$*;*
REPEAT : $i \neg i + 1$*;*
DO insert the service dependency link of $\bar{d}_i$ *into* $G(\overline{S})$*;*
UNTIL $i = h$*;*
RESET $i = 0$*;*
REPEAT : $i \neg i + 1$*;*
DO insert the USIE Branch of $\bar{l}_i$ *into* $G(\overline{S})$*;*
UNTIL $i = m$*;*
RETURN : $G(\overline{S})$

A composite service can be executed alternatively under specific configurations. Each configuration is composed of several supporting services related through service dependency relationships. We simply model a USIE configuration as a graph called USIE

Configuration graph. A USIE configuration graph consists of USIE atomic service entities and service dependency relationships. Each service entity in the configuration graph can be further elaborated by detailing the corresponding service's USIE model, in which case atomic service entities involved could be replaced by corresponding USIE graphs. Specifically, Table 4.2 describes the steps to construct a USIE Configuration graph given the supporting services and service dependencies.

According to Table 4.2, the construction of a USIE configuration graph involves three steps. Firstly, the service nodes are created in the graph. Secondly, the directed edges representing service dependencies are created; thirdly, the graph is augmented using suitable USIE branches.

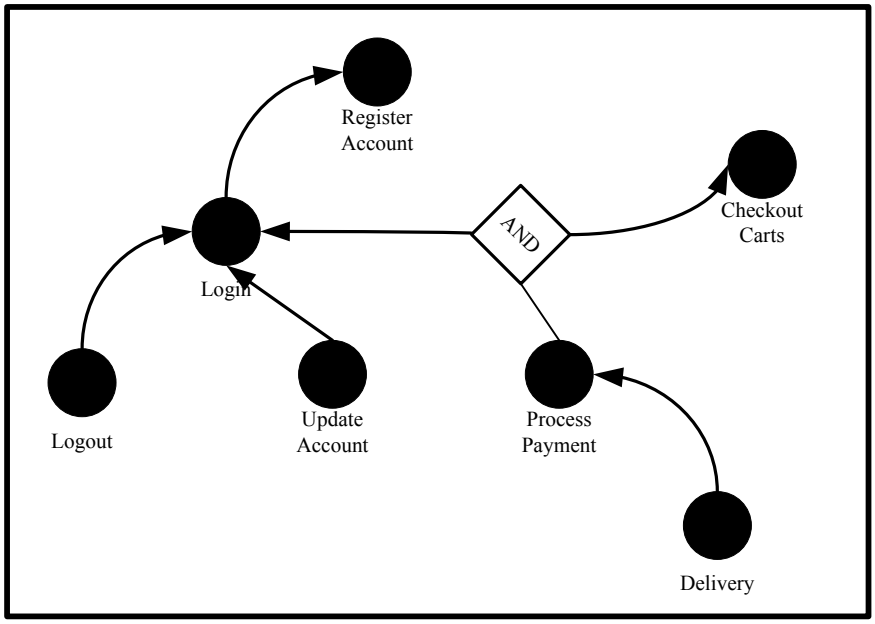

Figure 4.11. The USIE Configuration Graph for the Ordering Service

To illustrate the algorithm of Table 4.2, we consider as an example a configuration of the *Ordering Service* in the FS application. The configuration for a new user using the Ordering Service involves the following seven supporting services: *register, login, logout*, *update account, checkout carts, process payment* and *delivery*. Figure 4.11 illustrates the USIE configuration graph for the Ordering Service. The nodes of the graph correspond to the supporting services, while the directed edges express the dependencies among them.

As illustrated by Figure 4.11, the dependencies among the supporting services can be characterized as follows: service *register* is always active before the other supporting services; services *logout* and *update account* are always active after *login*; service *delivery* is active after service *process payment*; service *process payment* is active only after both service *checkout cart* and *login* are active. USIE branches are defined for service dependencies that have the same source service node. For instance, an "And" USIE branch is defined for the two outgoing dependencies from the service *process payment* to indicate the underlying logical relationship.

Each of the service nodes in the configuration graph can be further refined using their corresponding USIE representations. For instance, the *UpdateAccount* service in Figure 4.11 can be refined using the USIE model of Figure 4.10. Figure 4.12 shows the corresponding refined configuration graph.

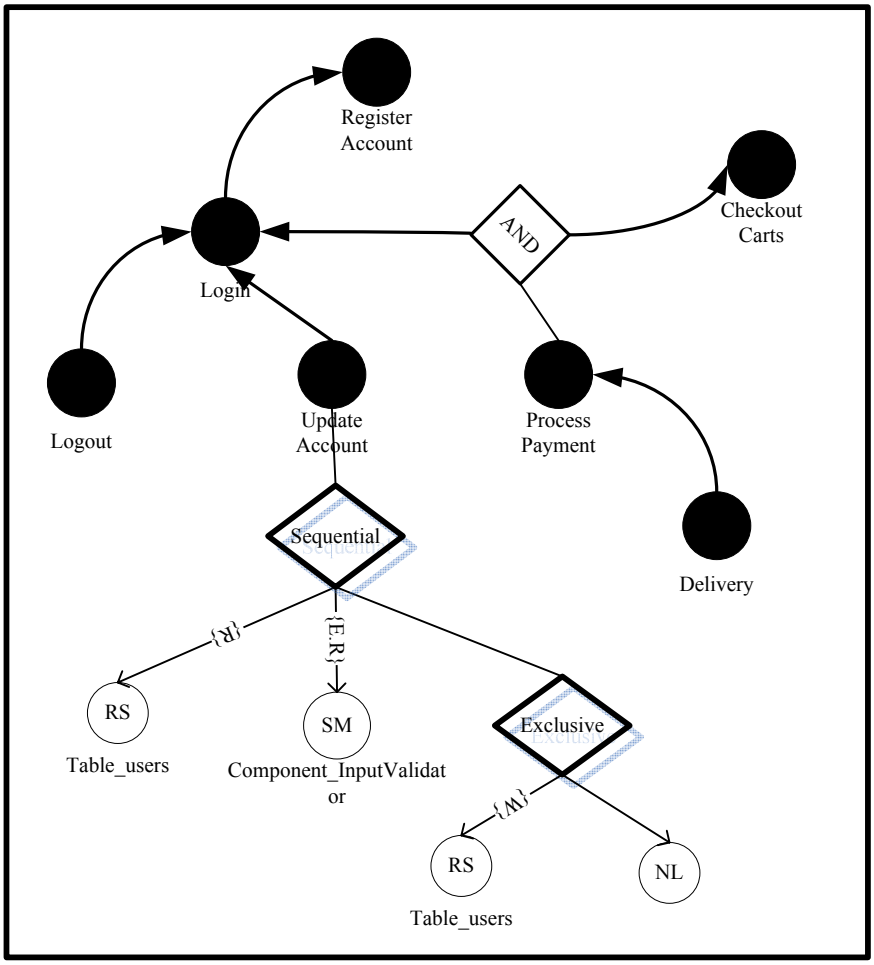

Figure 4.12. Refined USIE Configuration Graph for the Ordering Service

4.4.3 USIE Composite Service Graph

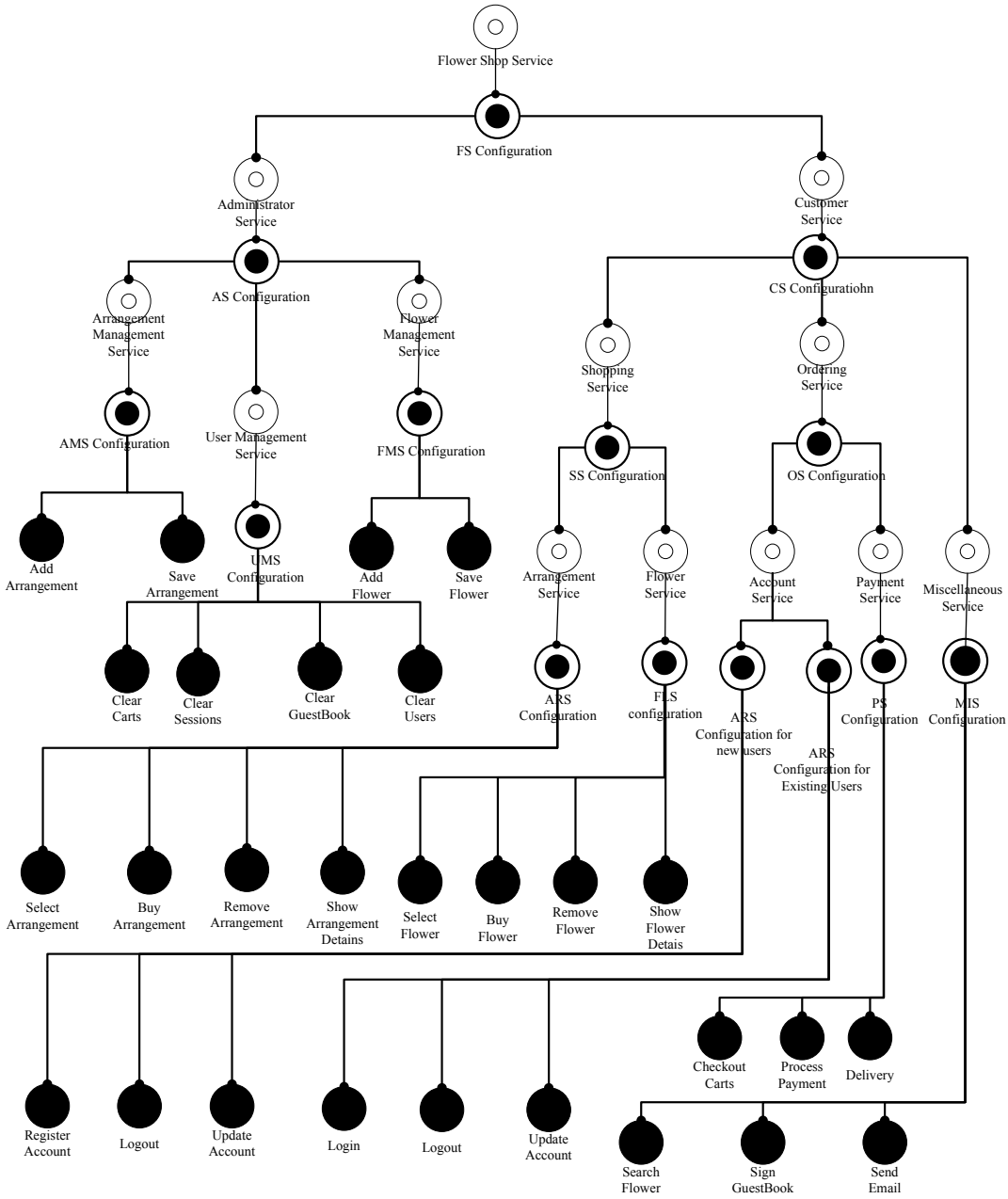

Figure 4.13. Composite Service Graph for FS Root Service

The purpose of a USIE composite service graph is to describe the service hierarchy underlying a specific composite service. It is represented as a tree in which the nodes correspond to either configuration or service entities and the edges depict the supporting relationships between the services and configurations entities.

Specifically, to construct a USIE composite service tree, we start by creating the root node representing the composite service itself. Then by proceeding iteratively, suitable nodes are created representing corresponding configurations and their supporting services. The edges linking services and configurations nodes in the composite service tree are expressed using USIE composition link elements. Each of the configuration and atomic service nodes can be further expanded by integrating corresponding USIE graphs.

As an example, Figure 4.13 depicts a composite service graph for the FS application. Specifically, the root node of the graph corresponds to a composite service representing the entire FS application. The rest of the FS services and configurations are represented in the graph iteratively based on the support relationships linking them. Particularly, in USIE composite service graph, a configuration node is defined in terms of its direct supporting services. As a result, a USIE composite service can be supported by USIE configurations at different hierarchical levels. As an example, in Figure 4.13, the *Ordering Service* involves a single configuration, which is directly supported by services *Account Service* and *Payment Service*. For instance, *Account Service* in its turn involves two different configurations, which in their turn are supported by other services, and so on and so forth.

4.5 Summary

In this chapter, we have motivated the need for security in service-oriented architecture, and introduced a new paradigm, the USIE model, to capture security features in this kind of architecture. The syntax of the USIE model is based on a set of simple graphical notations, which are easy to grasp and use. The semantic is based on generic software model introduced earlier in chapter 3. Using USIE abstraction, a software system can be represented as a collection of services structured hierarchically where the top service is a composite service representing the application itself. Furthermore, each of the services and configurations involved in the hierarchy can be described using a corresponding USIE graph. The model generation can be fully automated using the different algorithms introduced in this chapter.

USIE model can be used to define and derive various kinds of internal software metrics that can be used as basis for quantitative architecture-level security analysis of service-oriented software system. Such kind of analysis can help identifying and addressing potential security problems at the early stages of software development. We illustrate, in the next chapter, USIE-based security analysis by introducing number of relevant metrics and techniques.

Chapter 5

Security Analysis based on the USIE Paradigm

In this chapter, we present a framework for systematic security analysis of service-oriented architecture based on the USIE model. Our general approach for attackability analysis is to define and compute internal metrics based on internal security related attributes (e.g., complexity, coupling etc.). Attackability estimation is then carried out by interpreting the computed metrics according to the empirical security design hypotheses outlined in chapter 3 (see section 3.1.3) and repeated (here) as follows:

- ***Hypothesis 1:*** Security generally decreases as Service Complexity increases.
- ***Hypothesis 2:*** Security generally decreases as Service Coupling increases.
- ***Hypothesis 3:*** Security generally decreases as Service Excess Privilege increases.
- ***Hypothesis 4:*** Security generally increases as Service Mechanism Strength increases

During the analysis, internal metrics are computed and analyzed, and then security risks are identified and appropriate mitigation strategies are enacted. This process goes on iteratively until the overall attackability risk exposure is deemed acceptable. It is important, here, to note that (although desirable) we do not develop in this work explicit (prediction) models (e.g., equations, formulas) linking attackability and internal software metrics. Instead, we limit our scope to identifying the connections between specific external attackability and internal software metrics. Defining and validating explicit prediction models will require considerable amount of data collected from extensive empirical studies. In the rest of this chapter, we present for each kind of security-related internal attributes introduced earlier (in chapter 3) sample metrics and analysis techniques

to capture the corresponding analytical results. The analysis introduced can be conducted in an algorithmic way, which distinguishes our framework from existing architecture analysis techniques in the literature.

5.1 Service Complexity Analysis

Using USIE model, complexity of software services can be defined from various perspectives. In this section, we illustrate one such perspective based on the notion of *use patterns* for composite services.

5.1.1 Use Pattern Definition

A composite software service may be accessed or used in various ways to serve specific purposes. For instance, a customer may use a personal banking service through an online banking system to transfer money between accounts, while another customer may use the same service to pay bills. These various forms of *service uses* actually characterize how users interact with the system through the service. Normally, the uses of a composite service are composed of some basic forms, each specifying an access path to one of its supporting services. In this work, we term the basic forms of service uses as *service use patterns*. Identifying the use patterns of a composite software service helps to discover or estimate underlying security issues, for instance, by identifying possible misuse patterns. Specifically, the set of use patterns for a composite service corresponds to the union of the use patterns underlying the service configurations. Each use pattern involves a basic access route to a supporting service, and is usually characterized as one or many services executed in a certain order. In general, a basic form of use pattern of a composite service can be denoted as a sequence $< \bar{s}_1, \bar{s}_2, ... \bar{s}_n >$ where $\bar{s}_i$ represents an atomic service underlying the composite service and service $\bar{s}_i$ is always executed before service $\bar{s}_{i+1}$. If a composite service is composed of several other composite services, its use patterns can be derived by joining the use patterns of the contained composite services with respect to their service dependency relationships. For instance, the *Ordering* service of the FS application has one configuration supported by two composite services, namely *Payment*

service and *Account* service respectively (see Figure 4.13). The *Payment* service has also a dependency on the *Account* service. Since <*Login*> is a use pattern of the *Account* service and <*Checkout Carts*> is a use pattern of the *Payment* service, we can obtain a use pattern <*Login, Checkout Carts*> for the *Ordering* service by joining <*Login*> and <*Checkout Carts*> together. The *Checkout Carts* service comes after the *Login* service in the joined use pattern simply because the *Payment* service depends on the *Account* service. We can capture such combination of use patterns using a composition operation denoted $\times$, where given two use patterns $<\bar{s}_{11},\bar{s}_{12},...\bar{s}_{1i}>$ and $<\bar{s}_{21},\bar{s}_{22},...\bar{s}_{2j}>$, $<\bar{s}_{11},\bar{s}_{12},...\bar{s}_{1i}>\times<\bar{s}_{21},\bar{s}_{22},...\bar{s}_{2j}> = <\bar{s}_{11},\bar{s}_{12},...\bar{s}_{1i},\bar{s}_{21},\bar{s}_{22},...\bar{s}_{2j}>$.

5.1.2 Use Patterns Derivation

In practice, the USIE configuration graphs of a composite service can serve as the basis for deriving the use patterns of the service. As introduced earlier, the USIE configuration graph for a composite service specifies dependency relationships among corresponding supporting services. Simply by traversing the dependency relationships specified in a USIE configuration graph, we can identify the access paths for each of the supporting services. Table 5.1 presents an algorithm that defines a systematic way to derive use patterns from a USIE configuration graph.

In Table 5.1, two functions are defined to derive the use patterns for a configuration graph. The *AtomicService_UsePattern_Derivation* function takes as inputs a USIE configuration graph and an atomic service node, and returns the set of use patterns in the configuration graph for the given atomic service. The use patterns for each atomic service node are identified by recursively traversing the atomic service nodes contained in the configuration graph. The *USIE_UsePatterns_Derivation* function returns all of the use patterns involved in the given configuration graph by combining the set of use patterns associated with each of the atomic services involved.

Table 5.1. Algorithm for Use Pattern Derivation

FUNCTION : USIE_UsePatterns_Derivation (a USIE Configration Graph G)
RETURNS a set of use patterns UP(G);
INPUTS : a USIE Configration Graph $G =< N, E, <, L >$
where $N = \{n_1, n_2, ..., n_k\}, E = \{e_1, e_2, ..., e_h\}, L = \{l_1, l_2, ..., l_o\}$;
INITIALIZATION : $i = 0, UP(G) = \emptyset$;
REPEAT : $i \leftarrow i + 1$;
IF(n_i *is an AtomicService)*
DO $UP(G) = UP(G) \cup$ *AtomicService_Paths_Derivation(*G, n_i *);*
END IF
UNTIL $i = k$;
RETURN : UP(G)

FUNCTION : AtomicService_UsePattern_Derivation
(a USIE configuration graph $G =< N, E, <, L >$,
a USIE AtomicService node n where $n \in N$*)*
RETURNS a set of use patterns UP(G,n);
INPUTS : a USIE configuration graph $G =< N, E, <, L >$,
a USIE AtomicService node n where $n \in N$;
INITIALIZATION : a use pattern $p = (n), UP(G,n) = \emptyset$;
IF(n has no dependency in G)
DO $UP(G,n) = UP(G,n) \cup \{p\}$;
ELSE IF(n has a single dependency on node m in G)
DO $UP(G,n) = UP(G,n) \cup$ *(AtomicService_UsePattern_Derivation(*G, m*)*$\times\{p\}$*);*
ELSE IF(n has a "And" dependency on nodes $m_1, m_2, ... m_k$ *in G)*
DO $UP(G,n) = UP(G,n) \cup$
(AtomicService_UsePattern_Derivation(G, m_1 *)*
$\times ... \times$ *AtomicService_UsePattern_Derivation(*G, m_k *)*$\times\{p\}$*);*
ELSE IF(n has a "Or" dependency on nodes $m_1, m_2, ... m_k$ *in G)*
DO $UP(G,n) = UP(G,n) \cup$
(AtomicService_UsePattern_Derivation(U, m_1 *)*$\times\{p\}$*)*
$\cup ... \cup$ *(AtomicService_UsePattern_Derivation(*U, m_k *)*$\times\{p\}$*);*
ELSE
DO null;
RETURN : UP(G,n)

5.1.3 Use Pattern Metrics

Use patterns specify the alternative routes to access the supporting services for a composite service. As a matter of fact, the use patterns of a composite service represent one aspect of the inner structure of the service. Given a composite service, the more complex its use patterns are the more complex the service execution will be. We can therefore characterize quantitatively service complexity by deriving complexity metrics related to service use patterns.

Various complexity metrics can be derived from the use patterns of a composite service. Since the sequence length of each use pattern indicates actually the structural complexity for reaching an atomic service during the execution of a composite service, we define in the following, as an example, a simple complexity metric for a composite service based on the sequence length of use patterns.

Metric of Service Complexity: Given a composite service *cs* and its USIE configuration graph $U_{cs} =< N_{cs}, E_{cs,} <_{cs}, L_{cs} >$, the *Average Service Depth* (*ASD*) for U_{cs} is defined as

$$ASD(U_{cs}) = \frac{\sum_{n_i \in N_{cs}} NumOfServiceDependency(n_i) \times IsAtomicService(n_i)}{\sum_{n_i \in N_{cs}} IsAtomicService(n_i)}$$

Where:

$$IsAtomicService(n_i) = \begin{cases} 1, & if\ n_i\ is\ an\ atomic\ service\ node. \\ 0, & otherwise. \end{cases} \qquad \text{(Metric 7)}$$

$$NumOfServiceDependency(n_i) = The\ number\ of\ service\ nodes\ depending\ directly\ or\ indirectly\ on\ n_i.$$

The USIE configuration graph of a composite service explores the inner dependency relationships between atomic services involved. Intuitively, the more inner dependency relationships exist within a composite service, the more structurally complex the composite service ought to be. The *ASD* metric actually computes the average number of dependency relationships per atomic service node; therefore, high *ASD* values indicate

high degree of dependencies in the service. In other words, *ASD* metric can (intuitively) be used as a complexity metric for composite services. In addition, the theoretical validation of the *ASD* metric shows that it satisfies all the complexity axioms defined in our framework (see chapter 3). The detailed validation proofs are given in Appendix B.

5.2 Service Coupling Analysis

The concept of service coupling is used in this work to capture the amount of relationships between services. There are various kinds of relationships between software services. In this work, our interest lies in the relationships that may impact software security. Specifically in this section, we tackle coupling analysis from several perspectives including confidentiality, integrity and resource sharing. Confidentiality and integrity are two important security properties which are usually difficult to capture and analyze, and it is quite common that software systems are being developed without serious handling of confidentiality and integrity requirements. Our understanding of confidentiality and integrity is consistent with the work of Jacob who defines software security in terms of user-system interactions [32]. According to Jacob, security issues arise because of the occurrence of some user interactions with the system. A user-system interaction may affect system security either in isolation or by interfering with other interactions. Under the service-oriented framework, the interactions between users and system correspond to service executions. The concept of resource sharing, as introduced in Chapter 3, is also an important coupling relationship between software services. More specifically, as argued in Chapter 3, we can consider two software services to be coupled with each other if they share system components during service executions. In the rest of this section, we discuss basic resource sharing analysis followed by confidentiality and integrity analysis

5.2.1 Basic Resource Sharing

USIE abstractions describe service executions from security perspective using two kinds of system components, namely security mechanism components and resource components.

A software component is considered to be a security mechanism component if it provides or implements some form of protection during a service execution. A component is considered to be a resource component if it represents a form of system resources needing some protection. Various coupling metrics can be derived from the USIE abstractions of software services. In practice, many service relationships can be considered as service coupling relationships and their metrics can be defined initially based on intuitive understanding of the corresponding relationships. For illustrative purpose, we define as follows a sample service coupling metric based on the resource components involved in USIE models.

Metric of Resource Sharing: Given two services s_1, s_2 and their USIE configuration graphs $U_{s_1} = (N_{s_1}, E_{s_1}, <_{s_1}, L_{s_1})$ and $U_{s_2} = (N_{s_2}, E_{s_2}, <_{s_2}, L_{s_2})$, the *Ratio of Shared Resources (RSR)* between U_{s_1} and U_{s_2} is defined as

$$RSR(U_{s_1}, U_{s_2}) = \frac{cardinality(ResourceNodes(N_{s_1}) \cap ResourceNodes(N_{s_2}))}{cardinality(ResourceNodes(N_{s_1}) \cup ResourceNodes(N_{s_2}))}$$

Where: (Metric 8)

$ResourceNodes(N_i)$ *is the set of USIE Resource nodes contained in* N_i.

In our definition of service coupling, the amount of service coupling coincides with the amount of service sharing. The *RSR* metric actually captures the level of resource sharing between two given services by computing the ratio between the number of shared resources and the total number of resources involved in the services. Therefore, higher *RSR* metric values between services indicate higher degree of resource sharing, which in consequence indicate higher service coupling. In addition, the theoretical validation of the *RSR* metric shows that it satisfies all the coupling axioms defined in our framework. The detailed validation is presented in Appendix B.

5.2.2 Confidentiality Analysis

According to Jacob [32], *"confidentiality is about limiting how much one user can infer about another user's interaction with the system by making an interaction with the system themselves"* [32]. Hence, confidentiality is function of the interference occurring among interactions between different users with the system. Hence, evaluating this interference may serve in deriving possible metric of confidentiality. We denote by $Conf(I_A \rightarrow I_B)$ the confidentiality of I_A with respect to I_B, where I_A represents an interaction of a user A with the system, and I_B an interaction of a user B with the system. In other words, $Conf(I_A \rightarrow I_B)$ metrics how much B can discover about I_A via I_B. Under the service-oriented framework, I_A may represent a service execution carried by a service requestor A, and I_B may represent a service execution carried by a service requestor B.

Basically, confidentiality is related to information sharing; information leakage decreases confidentiality [32]. In [38], Kemmerer proposed that finding a pair of "*Read*" and "*Write*" operations on a shared resource could identify an information leakage channel between two processes. Accordingly, we can conduct confidentiality analysis based on the potential information leakage channels between services. More specifically, an information leakage channel exists whenever a service execution may involve reading from a resource that another service can write to, because the service requestor of the former service may deduce some information from the latter service execution, which decreases the confidentiality of the latter service.

Using the USIE atomic service graph, identification of information leakage channels between two atomic services is straightforward. A *"Read"* operation underlying an atomic service corresponds to a USIE operation edge containing *{R}* right in the corresponding USIE atomic service graph while a *"Write"* operation of an atomic service corresponds to a USIE operation edge containing *{W}* right in the corresponding USIE atomic service graph. We use the notation $(U_1 \rightarrow U_2, n)$ to express the fact that there is an Information Leakage Channel from USIE graph U_1 to USIE graph U_2 via the node n. Furthermore, the algorithm introduced in Table 5.2 can be used to derive automatically Information Leakage Channels between USIE atomic service graphs.

Table 5.2. Algorithm for ILC Derivation

FUNCTION : AtomicServices_ILC_Derivation
(a USIE AtomicService Graph $G(\bar{s}_1)$*, a USIE AtomicService Graph* $G(\bar{s}_2)$*)*
RETURNS a set of Information Leakage Channels $ILC(\bar{s}_1, \bar{s}_2)$*;*

INPUTS : a USIE AtomicService Graph $G(\bar{s}_1) =< N_1, E_1, <_1, L_1 >$*,*
a USIE AtomicService Graph $G(\bar{s}_2) =< N_2, E_2, <_2, L_2 >$*,*
where $N_1 = \{n_1, n_2, ..., n_k\}, E_1 = \{e_1, e_2, ..., e_h\}, L_1 = \{l_1, l_2, ..., l_p\}$*,*
$N_2 = \{n'_1, n'_2, ..., n'_q\}, E_1 = \{e'_1, e'_2, ..., e'_r\}, L_1 = \{l'_1, l'_2, ..., l'_t\}$*;*
INITIALIZATION : $i = 0, j = 0, ic = null, ILC(\bar{s}_1, \bar{s}_2) = \varnothing$*;*
REPEAT : $i \leftarrow i + 1$*;*
IF(n_i *is a USIE RSComponent)*
REPEAT : $j \leftarrow j + 1$*;*
IF(n_i *is identical to* n'_j*)*
IF($\{W\} \subseteq Rights(Incoming(n_i))$ *AND* $\{R\} \subseteq Rights(Incoming(n'_j))$*)*
DO $ic = (G(\bar{s}_1) \rightarrow G(\bar{s}_2), n_i)$*;*
DO $ILC(\bar{s}_1, \bar{s}_2) = ILC(\bar{s}_1, \bar{s}_2) \cup \{ic\}$*;*
END IF
END IF
UNTIL $j = q$*;*
END IF
UNTIL $i = k$*;*
RETURN : $ILC(\bar{s}_1, \bar{s}_2)$

As expressed by Table 5.2, the information leakage channels between two USIE atomic service graphs are identified by traversing the resource components that are shared by both graphs. An information leakage channel exists from service $\bar{s}_1$ to service $\bar{s}_2$ when there is a read action to a component on the graph of service $\bar{s}_1$ and a write action to the same component on the graph of service $\bar{s}_2$. The set of information channels between two graphs are the union of the individual information leakage channels.

Confidentiality analysis in our context can be conducted once the information leakage channels between services are identified. Various forms of confidentiality metrics may be developed depending on the specific confidentiality perspective to be analyzed. In this work, we introduce in the following a sample metric that can be used to measure

quantitatively $Conf(\bar{s}_1 \rightarrow \bar{s}_2)$ based on the information leakage channels derived for two services.

Metric of Confidentiality: Given software services $\bar{s}_1$, $\bar{s}_2$ and the set of information leakage channels $ILC(\bar{s}_1, \bar{s}_2)$, $Conf(\bar{s}_1 \rightarrow \bar{s}_2)$ can be computed as follows:

$$Conf(\bar{s}_1 \rightarrow \bar{s}_2) = \frac{1}{1 + count(ILC(\bar{s}_1, \bar{s}_2))} \qquad \text{(Metric 9)}$$

As defined earlier in this section, $Conf(\bar{s}_1 \rightarrow \bar{s}_2)$ measures how much the service requester of $\bar{s}_2$ can discover about the service $\bar{s}_1$ via the execution of $\bar{s}_2$. Intuitively, the more information leakage channels exist between services, the less confidentiality is preserved. Accordingly, the *Conf* metric may be used as one metric of $Conf(\bar{s}_1 \rightarrow \bar{s}_2)$ since it returns a normalized value (in [0,1]) that indicates the amount of information leakage channels between service $\bar{s}_1$ and $\bar{s}_2$. For instance, the *Conf* metric will return 1 indicating that $Conf(\bar{s}_1 \rightarrow \bar{s}_2)$ is maximum if there is no information leakage channels between two services $\bar{s}_1$ and $\bar{s}_2$.

Example of Confidentiality Analysis: We analyze information leakage between two atomic services of the FS application, namely the *Checkout cart* service and the *BuyFlower* service. The USIE graphs of the *Checkout cart* service and the *BuyFlower* service are shown in Figure 5.1. The execution of the *BuyFlower* service consists of three sequential USIE operations accessing three different FS resources, namely *Table_flowers*, *Table_flowerCart*, and *Table_carts*. The *Checkout cart* service involves more operations than the *BuyFlower* service. Its execution starts with an operation on the security mechanism *Component_CookieManager* that checks some user credential. If the checking fails, the execution stops, otherwise five operations are invoked in sequence to perform the checkout service as illustrated by Figure 5.1 (b). As mentioned earlier, confidentiality breach between two services can be established by identifying underlying information leakage channels. Using USIE model and the algorithm of Table 5.2, we can derive automatically information leakage channels between the atomic services *"CheckoutCart"*

and *"BuyFlower*. The information leakage channels between these two services are illustrated in Figure 5.2 using dashed directed arcs.

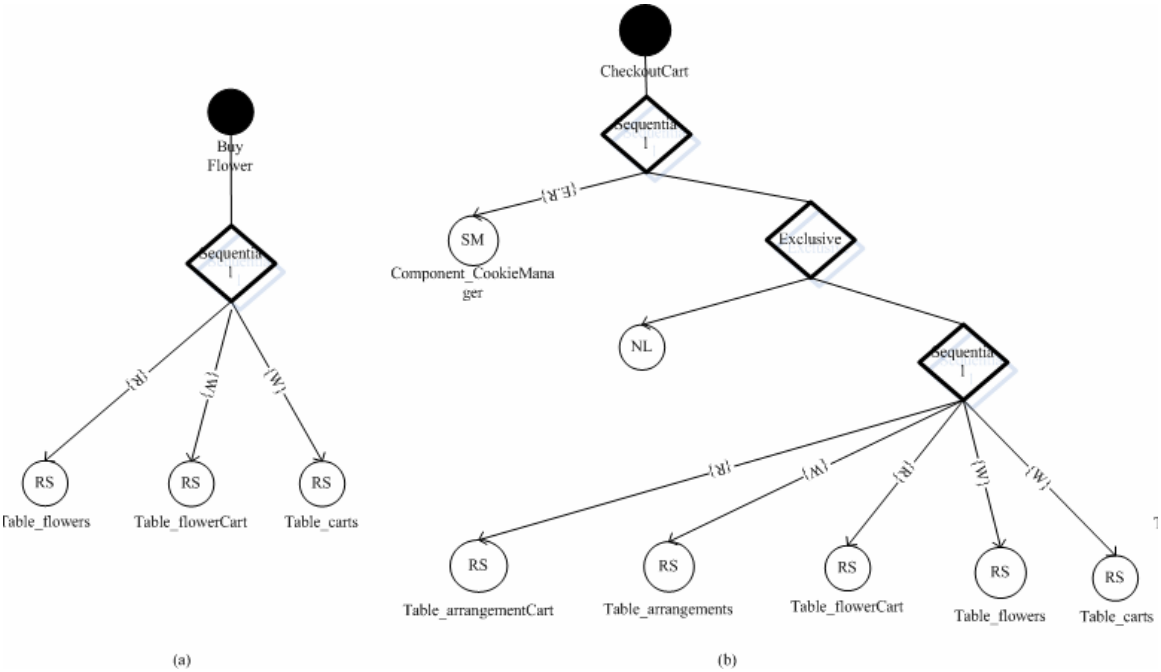

Figure 5.1 USIE Graphs for (a) *BuyFlower* Service and (b) *CheckoutCart* Service.

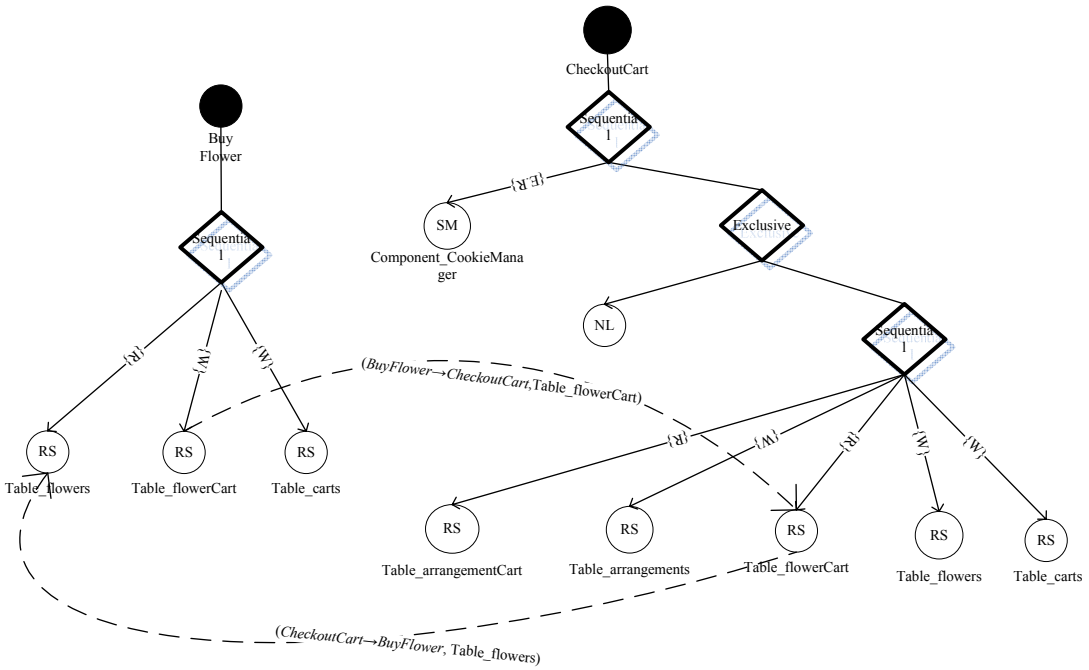

Figure 5.2. Information leakage channels between services *BuyFlower* and *CheckoutCart*

Furthermore, we can quantify the confidentiality between the *"CheckoutCart"* service and the *"BuyFlower"* service using the *Conf* metric defined earlier as follows:

$$Conf(\overline{CheckoutCart} \rightarrow \overline{BuyFlower}) = \frac{1}{1+count(ILC(\overline{CheckoutCart}, \overline{BuyFlower}))} = 0.5$$

$$Conf(\overline{BuyFlower} \rightarrow \overline{CheckoutCart}) = \frac{1}{1+count(ILC(\overline{BuyFlower}, \overline{CheckoutCart}))} = 0.5$$

According to the above *Conf* metrics, there is a potential risk of breach of confidentiality between the two services. Software architects can therefore use the measurement results to determine whether the service design needs to be adjusted to reduce confidentiality risks between the two services.

5.2.3 Integrity Analysis

Jacob defines integrity as the process of *"ensuring that data are altered only in an approved fashion."* In this section we focus on the integrity analysis of software services. In our context, we consider that the integrity of a service execution (a user-system interaction) is preserved as long as all the data entities involved are altered without being interfered by other service executions. However, in software systems, data entities or resources are in many cases shared by multiple services and may be accessed in multi-threading environment. Concurrent access to data entities provides flexibility and simplicity in software development, but also creates potential security problems. Hence, careful design of concurrency mechanisms is essential in preserving service integrity. We denote $Integ(I_A \rightarrow I_B)$ as the integrity of a user system interaction I_A with respect to a user system interaction I_B. In other words, $Integ(I_A \rightarrow I_B)$ measures how much I_B can affect I_A via the execution of I_B.

Basically, integrity is related to information modification [32]. Accordingly, we need to determine the modification channels existing between software services when conducting service integrity analysis. In our context, a modification channel from a service to another service exists typically when the former service carries a *"Write"*

operation on some shared data entities or resources. Using USIE models of software services, we can identify systematically such modification channels between services. Using USIE notations, we denote a modification channel as $(U_1 - U_2, n)$ to represent that there exists a modification channel from USIE model U_1 to U_2 via node n. Table 5.3 introduces a systematic way to derive modification channels between a pair of USIE atomic service graphs.

Table 5.3. Algorithm for MC Derivation

FUNCTION : AtomicServices_MC_Derivation
(a USIE AtomicService Graph $G(\bar{s}_1)$*, a USIE AtomicService Graph* $G(\bar{s}_2)$*)*
RETURNS a set of Modification Channels $MC(\bar{s}_1, \bar{s}_2)$*;*

INPUTS : a USIE AtomicService Graph $G(\bar{s}_1) = < N_1, E_1, <_1, L_1 >$*,*
a USIE AtomicService Graph $G(\bar{s}_2) = < N_2, E_2, <_2, L_2 >$
where $N_1 = \{n_1, n_2, ..., n_k\}, E_1 = \{e_1, e_2, ..., e_h\}, L_1 = \{l_1, l_2, ..., l_p\}$*,*
$N_2 = \{n'_1, n'_2, ..., n'_q\}, E_1 = \{e'_1, e'_2, ..., e'_r\}, L_1 = \{l'_1, l'_2, ..., l'_t\}$*;*
INITIALIZATION : $i = 0, j = 0, mc = null, MC(\bar{s}_1, \bar{s}_2) = \varnothing$*;*
REPEAT : $i \leftarrow i + 1$*;*
IF(n_i *is a USIE RSComponent)*
REPEAT : $j \leftarrow j + 1$*;*
IF(n_i *is identical to* n'_j*)*
IF($\{W\} \subseteq Rights(Incoming(n'_j))$*)*
DO $mc = (G(\bar{s}_1) - G(\bar{s}_2), n_i)$*;*
DO $MC(\bar{s}_1, \bar{s}_2) = MC(\bar{s}_1, \bar{s}_2) \cup \{mc\}$*;*
END IF
END IF
UNTIL $j = q$*;*
END IF
UNTIL $i = k$*;*
RETURN : $MC(\bar{s}_1, \bar{s}_2)$

As expressed by Table 5.3, the modification channels between two USIE atomic service graphs are identified by traversing their shared resource components. A modification channel exists through a shared component from service $\bar{s}_1$ to service $\bar{s}_2$

when there is a write action to the component in service $\bar{s}_2$. The set of modification channels between two graphs are the union of the individual modification channels.

Based on the notion of modification channel, we can develop various forms of metrics to capture quantitatively $Integ(\bar{s}_1 \rightarrow \bar{s}_2)$ in our context. For instance, metric 9 defined in the following is based on the number of modification channels:

Metric of Integrity: Given software services $\bar{s}_1$, $\bar{s}_2$ and the set of modification channels $MC(\bar{s}_1, \bar{s}_2)$, $Integ(\bar{s}_1 \rightarrow \bar{s}_2)$ can be computed as follows:

$$Integ(\bar{s}_1 \rightarrow \bar{s}_2) = \frac{1}{1 + count(MC(\bar{s}_1, \bar{s}_2))} \qquad \text{(Metric 9)}$$

The measurement of $Integ(\bar{s}_1 \rightarrow \bar{s}_2)$ evaluates how much a requester of service $\bar{s}_1$ can affect the execution of service $\bar{s}_2$ via the execution of $\bar{s}_1$. Intuitively, the more modification channels exist between services $\bar{s}_1$ and $\bar{s}_2$, the less $Integ(\bar{s}_1 \rightarrow \bar{s}_2)$ is preserved. The *Integ* metric can be used to compute $Integ(\bar{s}_1 \rightarrow \bar{s}_2)$ since it returns a normalized value that indicates the amount of the modification channels between service $\bar{s}_1$ and $\bar{s}_2$. For instance, if there is no modification channel between two services $\bar{s}_1$ and $\bar{s}_2$, the *Integ* metric will return 1 indicating that $Integ(\bar{s}_1 \rightarrow \bar{s}_2)$ is maximum.

Example of Integrity Analysis: As suggested above, integrity breach between two services can be studied by identifying corresponding modification channels. As an example, using USIE model and the algorithm of Table 5.3, we can derive modification channels between the atomic service *"CheckoutCart"* and *"BuyFlower"*. Figure 5.3 depicts the modification channels between both services. In the figure, the modification channels between the two services are marked using dashed directed arcs. Furthermore, we can quantify the integrity between the *"CheckoutCart"* service and the *"BuyFlower"* service using the *Integ* metric defined earlier as follows:

$$Integ(\overline{CheckoutCart} \rightarrow \overline{BuyFlower}) = \frac{1}{1 + count(MC(\overline{CheckoutCart}, \overline{BuyFlower}))} = 0.333$$

$$Integ(\overline{BuyFlower} \rightarrow \overline{CheckoutCart}) = \frac{1}{1+count(MC(\overline{BuyFlower}, \overline{CheckoutCart}))} = 0.333$$

According to the above *Integ* metrics, there is a potential risk of breach of integrity between the two services. Software architects can use the measurement results to determine whether the service design needs to be adjusted to reduce integrity risk between the two services.

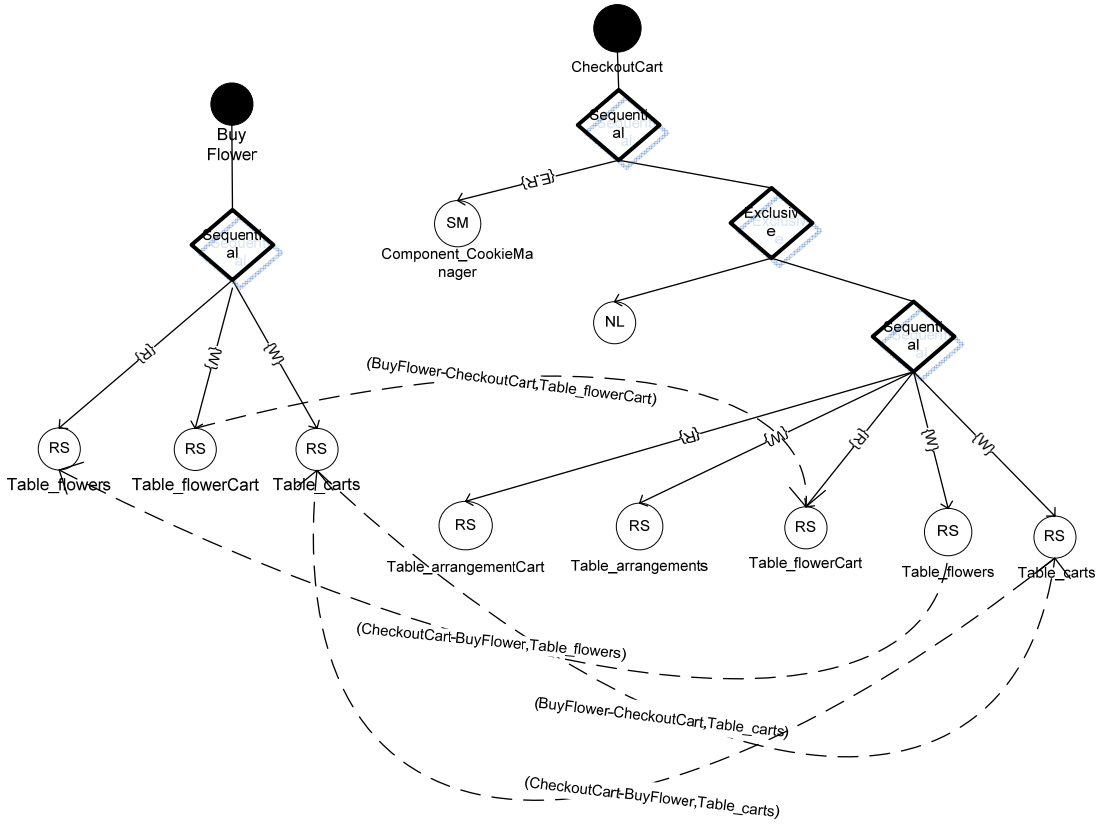

Figure 5.3. Modification channels between services BuyFlower and CheckoutCart

5.2.4 Discussion

Online FS services can be accessed concurrently by multiple users. In principle, software architects should conduct service coupling analysis for each pair of services to mitigate the possibility of unauthorized information transfer or alteration between different users.

By defining and applying internal coupling metrics like the RSR metric defined above, architect can infer and mitigate the amount of attackability exposure related to service coupling by restructuring accordingly the software architecture

It is important to mention, here, that although the two metrics, *Conf* and *Integ*, introduced above intuitively may be considered as coupling metrics, they are not because they failed our coupling properties as illustrated in Appendix B. These represent examples of metrics which intuitively could be linked to specific software attribute, but in principle should not be. In the above cases, by revising the metrics definition to address the failed properties, we should be able to address the shortcomings.

It is important to note that a negative side-effect of the service coupling metrics introduced in this section is that it could push the software designer and architects towards reducing data sharing between services in order to improve service security. In this work, we don't intend to claim that re-use and data sharing shall be prevented in software design to achieve higher security, we simply want to show that re-use and data sharing have impact on software security attribute. As discussed in [77], there will always be a need for trade-off between security quality and some other software concerns such as usability and flexibility etc. It is the role of the software architect to conduct proper trade-off analysis in order to decide how to balance appropriately the impact of important but conflicting software attributes.

5.3 Service Excess Privilege Analysis

In a software system, users are assigned some privileges according to the underlying security policy. According to the *least privilege* principle, a user should be granted only the minimum number of privileges he needs to accomplish a job. However, in reality, the implementation and configuration of software systems usually grant unnecessary privileges to users for various reasons mainly related to careless design and over simplification. The concept of *Excess Privilege* originally introduced by Terry A. Smith [62] refers to the amount of unnecessary privileges assigned to a user given a specific user domain and task. As mentioned in Chapter 3, privileges are related to services. From the perspective of software products, the actual privileges granted to users can be derived

from the interactions between service requesters and services. The difference between the *actual privileges set* granted and the *minimum set of privileges needed* corresponds to the *Excess Privileges*.

5.3.1 Privileges Derivation

Table 5.4. Algorithm for Privileges Derivation

FUNCTION : AtomicService_Privileges_Derivation(a USIE AtomicService Graph $G(\bar{s})$*)*
RETURNS a set of privileges $P(\bar{s})$*;*

INPUTS : a USIE AtomicService Graph $G(\bar{s}) =< N, E, <, L >$
where $N = \{n_1, n_2, ..., n_k\}, E = \{e_1, e_2, ..., e_h\}, L = \{l_1, l_2, ..., l_o\}$*;*
INITIALIZATION : $i = 0, P(\bar{s}) = \varnothing$*;*
DO $P(\bar{s}) = P(\bar{s}) \cup \{(ServiceRequestor(\bar{s}), \bar{s}, Rights(e_i))\}$*;*
REPEAT : $i \leftarrow i + 1$*;*
IF(e_i *is a USIEOperation)*
DO $P(\bar{s}) = P(\bar{s}) \cup \{(ServiceRequestor(\bar{s}), Target(e_i), Rights(e_i))\}$*;*
END IF
UNTIL $i = h$*;*
RETURN : $P(\bar{s})$

Based on the USIE model, we can derive the actual privileges associated with a software service. Specifically, in the USIE atomic service graph, an atomic service consists of a sequence of USIE operations among software components. Each USIE operation actually maps into a privilege that a service requestor can exercise on the target entity of the operation. In our framework, we use *ServiceRequestor* as the common notation to represent a service requestor. For instance, if a service $\bar{s}$ involves a USIE operation that specifies a *read* action for a target component *c* (shown as a USIE operation edge with the right attribute *{R}* in the corresponding USIE model), we can derive a corresponding privilege for service $\bar{s}$ as $(ServiceRequestor(\bar{s}), c, \{R\})$. To represent the execution privilege that an atomic service requestor has for the service itself, we associate each atomic service $\bar{s}$ with a specific privilege $(ServiceRequestor(\bar{s}), \bar{s}, \{E\})$. Table 5.4

defines in the following a systematic way to derive actual privileges from a USIE atomic service graph.

According to Table 5.4, the privileges associated with an atomic service can be derived from its corresponding USIE graph in two steps. Firstly, an execution privilege on the service itself is derived. Secondly, the privileges involved in the execution of the service can be identified by deriving privileges from each of the USIE operations involved in the corresponding USIE graph. Accordingly, the set of privileges of a composite service which by definition involves several supporting services can be derived as the union of the privileges sets associated with its supporting services.

5.3.2 Excess Privilege Metrics

By deriving actual privileges sets from USIE graphs of software services, many meaningful metrics can be defined and used to conduct privileges analysis with respect to specific objectives. As mentioned previously, excess privileges can be expressed as the difference between the actual privileges and the required privileges [62]. Accordingly, we present in the following a sample metric that can be used as a quantitative criterion to assess the excess privileges related to a service.

Metric of Excess Privilege: Given a software service $\bar{s}$, its actual privileges set $ActualPrivileges(\bar{s})$ and its least privileges set $LeastPrivileges(\bar{s})$, the *Excess Privilege Count* of service $\bar{s}$, denoted by $EPC(\bar{s})$, is given by

$$EPC(\bar{s}) = count(ActualPrivileges(\bar{s}) - LeastPrivileges(\bar{s})) \quad \text{(Metric 10)}$$

The *EPC* Metric can be used as an indicator for the measurement of service excess privileges since it measures the difference between actual privileges and required privileges. We also show in appendix B that the *EPC* metrics fulfill all the excess privilege metrics axioms defined in our framework.

5.3.3 Example of Service Excess Privilege Analysis

In principle, software users should be granted only necessary privileges to accomplish their jobs (also referred to as least privilege principle). However, in practice, users may be granted unnecessary privileges that can be potentially misused by malicious users due to careless design or oversimplification,. In this context, it is necessary to identify the actual privileges associated with each service and adjust service design by minimizing as much as possible unnecessary privileges. When designing the services of the FS application, software architects should conduct privilege analysis on each of the system services to make sure that unnecessary privileges are not assigned to system users. As an example, we take the *Payment Service* of the FS application to illustrate privilege analysis using USIE graphs. The *Payment Service* is a composite service containing a single configuration in terms of its supporting atomic services, which are *CheckoutCart*, *ProcessPayment,* and *Delivery*. We describe the service using its USIE configuration graph as illustrated in Figure 5.4. Specifically, the configuration graph consists of three atomic service nodes corresponding to the supporting services of the *Payment Service*. These atomic service nodes are connected by service dependency arcs. Specifically, the *Delivery* service depends on the *ProcessPayment* service and the *ProcessPayment* service depends on the *CheckoutCart* service. Each of the atomic service nodes is further refined by its corresponding USIE atomic service graph.

As shown in Figure 5.4, three atomic services (i.e., *CheckoutCart, ProcessPayment, Delivery*) with specific dependency relationships are involved in the *Payment Service*, each characterized by specific USIE operations. Using the algorithm of Table 5.4, we can derive systematically the actual privileges underlying the service from its USIE graph. Table 5.5 lists the actual privileges derived for the *Payment Service* from Figure 5.4.

The information collected in Table 5.5 is significant for software architects and developers in the sense that the excess privilege of the *Payment Service* can be evaluated objectively if the corresponding required least privileges are precisely defined. For instance, one of the rules of the security policy defined for the FS application states that *"A user cannot change sessions during the payment process."* Consequently the privilege $(ServiceRequestor(\overline{Payment}), Table\_sessions, \{W\})$ can be considered to be an

excess privilege underlying the *Payment* service since it allows the service requestor to modify the resource storing session information. Suppose that this is the only excess privilege identified for the *Payment Service,* we can compute the excess privileges metric defined earlier for the *Payment Service* as $EPC(\bar{s}) = 1$. Generally speaking, software Architects can use the *EPC* metrics of software services to guide and adjust service designs with respect to specific security policy requirement.

Table 5.5. Actual Privileges of the Payment Service

No.	Actual Privileges of service $\overline{Payment}$
1	$(ServiceRequestor(\overline{Payment}), \overline{CheckoutCart}, \{E\})$
2	$(ServiceRequestor(\overline{Payment}), \overline{ProcessPayment}, \{E\})$
3	$(ServiceRequestor(\overline{Payment}), \overline{Delivery}, \{E\})$
4	$(ServiceRequestor(\overline{Payment}), Component\_CookieManager, \{E,R\})$
5	$(ServiceRequestor(\overline{Payment}), Table\_arrangementCart, \{R\})$
6	$(ServiceRequestor(\overline{Payment}), Table\_arrangements, \{W\})$
7	$(ServiceRequestor(\overline{Payment}), Table\_flowers, \{W\})$
8	$(ServiceRequestor(\overline{Payment}), Table\_carts, \{W\})$
9	$(ServiceRequestor(\overline{Payment}), Table\_flowerCart, \{R\})$
10	$(ServiceRequestor(\overline{Payment}), Table\_sessions, \{R,W\})$
11	$(ServiceRequestor(\overline{Payment}), Table\_users, \{W\})$
12	$(ServiceRequestor(\overline{Payment}), Table\_arrangementCart, \{W\})$
13	$(ServiceRequestor(\overline{Payment}), Table\_flowerCart, \{W\})$
14	$(ServiceRequestor(\overline{Payment}), Table\_carts, \{W\})$

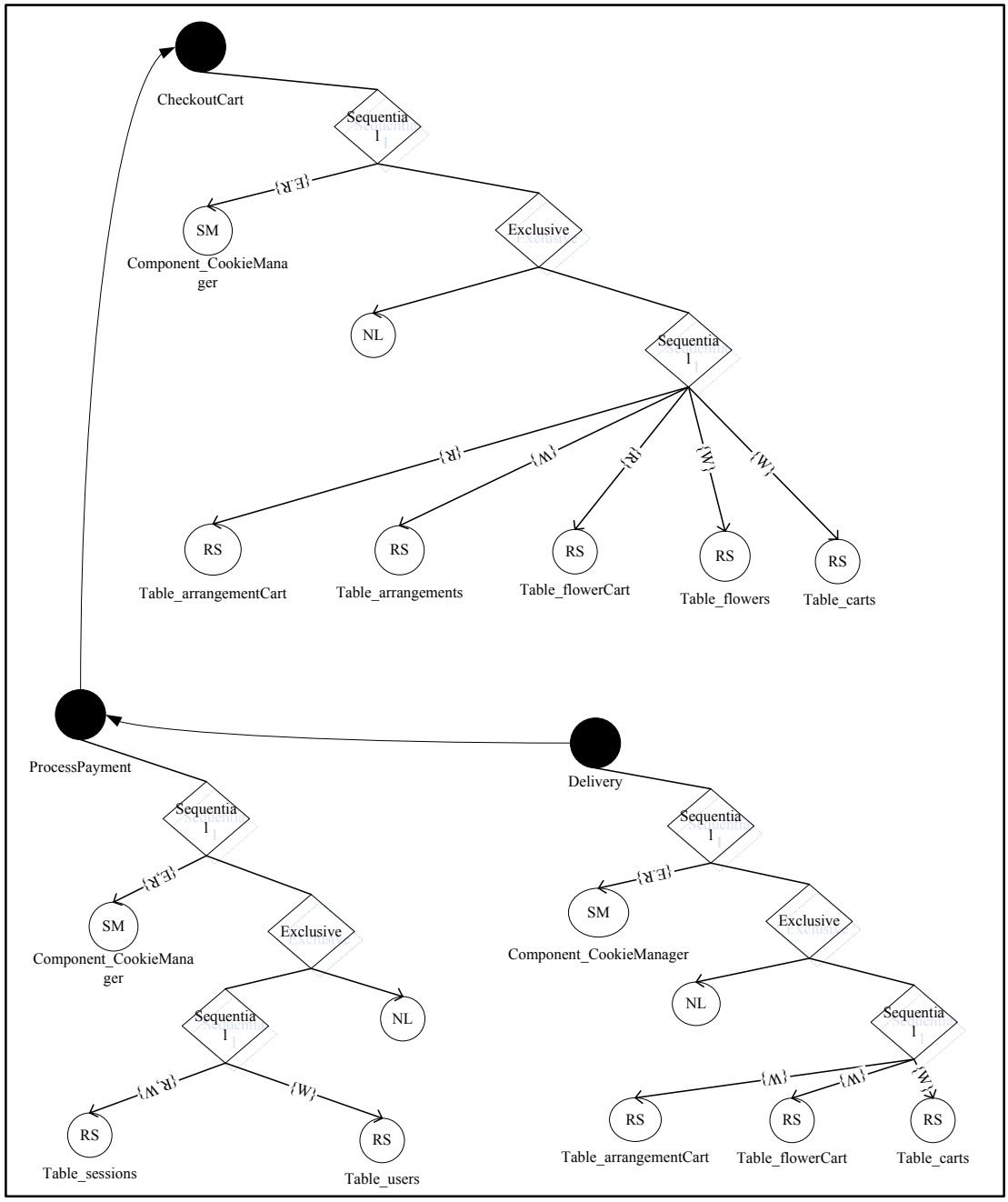

Figure 5.4. The USIE Configuration Graph of the *Payment* Service

5.4 Service Mechanism Strength Analysis

In software applications, privileges carried by software services are normally restricted or protected through appropriate security mechanisms, and some privileges may deserve more restriction than others. A security mechanism can be, for instance, an encryption

program, a password-checking program, or other kinds of security control or protection units. Generally, security mechanisms could be viewed as special kinds of modules that may involve one or more software components and provide particular security related services. In this framework, we are primarily interested in the security analysis of software services. We therefore focus on the strength of the security mechanisms involved in the protection of a particular software service. Specifically, the mechanism strength of a service is conveyed by the protection strength of the privileges associated with it. In other words, knowing how the privileges granted through a service are protected is the basis of the analysis of service mechanism strength. In this section, we pair privilege and mechanism in a new construct that is used as basis to analyze service mechanism strength.

5.4.1 Privilege-Mechanism Pair

Table 5.6. Examples of PMPs

PMP = (Privilege, Mechanism Composition)	***Interpretation***
$((Service\,\mathrm{Re}questor(\bar{s}),\ \bar{s},\ \{E\}),\ sm)$	Privilege *(ServiceRequestor($\bar{s}$), $\bar{s}$, {E})* is protected by mechanism sm.
$((Service\,\mathrm{Re}questor(\bar{s}),\ \bar{s},\ \{E\}),\ null)$	Privilege *(ServiceRequestor($\bar{s}$), $\bar{s}$, {E})* has no protection.
$((Service\,\mathrm{Re}questor(\bar{s}),\ \bar{s},\ \{E\}),\ sm_1 \wedge sm_2)$	Privilege *(ServiceRequestor($\bar{s}$), $\bar{s}$, {E})* is protected both by mechanisms sm_1 and sm_2.
$((Service\,\mathrm{Re}questor(\bar{s}),\ \bar{s},\ \{E\}),\ sm_1 \vee sm_2)$	Privilege *(ServiceRequestor($\bar{s}$), $\bar{s}$, {E})* is protected by mechanisms sm_1 or sm_2.

As shown in section 5.3, the set of privileges associated with a service can be derived automatically from the corresponding USIE graphs. Actually, the link between the privileges and relevant security mechanisms can also be inferred from the USIE graphs. In this work, we use a new construct named Privilege-Mechanism Pair (PMP) to derive and express such information. Specifically, we represent a PMP as a pair (Privilege, Mechanism Composition) in which Privilege corresponds to the expression of the privilege using the notation given in section 3.2.2 and the Mechanism Composition captures the list of security mechanisms protecting the privilege with their logical

relationships. Table 5.6 illustrates some examples of PMPs and corresponding interpretations.

5.4.2 Privilege-Mechanism Pair Derivation

Table 5.7. Algorithm for PMP Derivation

FUNCTION : Service_PMP_Derivation (a USIE Configuration Graph $G(\bar{s})$*)*
RETURNS a set of Privilege - Mechanism Pair $PMP(\bar{s})$*;*

INPUTS : a USIE Configuration Graph $G(\bar{s}) = < N, E, <, L >$
where $N = \{n_1, n_2, ..., n_k\}, E = \{e_1, e_2, ..., e_h\}, L = \{l_1, l_2, ..., l_p\}$*;*
INITIALIZATION : $i = 0, P = \varnothing, p = null, mc = null, PMP(\bar{s}) = \varnothing$*;*
REPEAT : $i \leftarrow i + 1$*;*
IF(n_i *is AtomicService node)*
DO $P = AtomicService\_Privileges\_Derivation(G(n_i))$*;*
END IF
FOR EACH : $p \in P$
IF(n_i *has no dependencies)*
DO $mc = mechanisms(n_i)$*;*
ELSE IF(n_i *has "And" dependencies on service nodes* $n_1', n_{2...}' n_m'$*)*
DO $mc = mechanisms(n_i) \wedge mechanisms(n_1') \wedge ... \wedge mechanisms(n_m')$*;*
ELSE IF(n_i *has "Or" dependencies on service nodes* $n_1', n_{2...}' n_m'$*)*
DO $mc = mechanisms(n_i) \wedge (mechanisms(n_1') \vee ... \vee mechanisms(n_m'))$*;*
END IF
DO $PMP(\bar{s}) = PMP(\bar{s}) \cup \{(p, mc)\}$*;*
END FOR EACH
UNTIL : $i = k$*;*
RETURN : $PMP(\bar{s})$

In the USIE graphs, the mechanism composition information is implicitly illustrated by USIE branch elements. For instance, the branch elements for USIE operations (*i.e., sequential, exclusive,* and *concurrent*) indicate the execution order of the security mechanism components associated with corresponding operations. The branch elements for service dependencies (*i.e., and, or*) indicate the logical relationships between security

mechanism components involved in the service executions. Accordingly, we can derive systematically the PMPs of a software service using corresponding USIE graphs.

Table 5.7 defines the procedure to derive the set of PMPs from the USIE configuration graph of a software service. According to the algorithm of Table 5.7, the derivation of PMPs from a USIE configuration graph involves privileges derivation and computation of security mechanism compositions for each of the derived privileges. Specifically, the USIE branch elements are used as guide to derive the security mechanism compositions for each privilege.

5.4.3 Mechanism Strength Metrics Definition

Based on the PMPs of software services, a variety of metrics can be developed to assess the security protection strength of software services. For instance, we can evaluate the protection strengths of service privileges with respect to the "*Weakest Link"* principle discussed earlier [60]. According to the "*Weakest Link"* principle, the security level of a system protected by a collection of security mechanisms is determined by the weakest protection mechanism. Based on this principle, we define in the following Metric 11 as an example of Mechanism Strength metric related to the privileges underlying a software service.

Metric of Mechanism Strength: Given a software service $\bar{s}$ and the set of Privilege-Mechanism Pairs $PMP(\bar{s})$, the *Privilege Mechanism Strength* of a privilege p_r in $\bar{s}$, denoted by $PMS(p_r, \bar{s})$ is given by

$$PMS(p_r, \bar{s}) = \underset{i \in PMP(\bar{s}) \ \& \ p_r \in \Pr ivileges(i)}{Min} \{MechanismStrength(i)\} \qquad \text{(Metric 11)}$$

Where $\Pr ivileges(i)$ represents the privileges involved in the PMP *i.*

$MechanismStrength(i)$ represents the strength of the mechanism composition involved in the PMP *i.*

To compare the strength of a composition of various security mechanisms, we need to normalize the individual metrics involved. One way to achieve that is by assigning

normalized weights to each of the security mechanisms involved and computing a global value based on the composition rules. The value of a mechanism weight is mechanism-specific, and could be provided by corresponding vendor or computed using specific evaluation techniques. For instance the rating technique proposed by Eloff for security mechanisms such as cryptographic primitives or password mechanisms could be used [17]. The theoretical validation for the PMS metric is presented in appendix B.

5.4.4 Example of Service Mechanism Strength Analysis

Knowing how privileges are protected is critical for the FS application since its underlying services are delivered online in an open environment where intrusions and hacking activities happen all the time. Software architects should be certain that appropriate security mechanisms are designed in FS services to protect each of the privileges that need protection. As an example, we demonstrate service mechanism strength analysis using the USIE configuration graph of the *Payment Service* (See Figure 5.4). Table 5.8 presents the PMP units derived from Figure 5.4 using the algorithm of Table 5.7.

Table 5.8. PMP Units Derived From Payment Service

No.	***PMP Units***
1	$((ServiceRequestor(\overline{Payment}), \overline{CheckoutCart}, \{E\}), null)$
2	$((ServiceRequestor(\overline{Payment}), \overline{ProcessPayment}, \{E\}), Component\_CookieManager)$
3	$((ServiceRequestor(\overline{Payment}), \overline{Delivery}, \{E\}), Component\_CookieManager)$
4	$((ServiceRequestor(\overline{Payment}), Component\_CookieManager, \{E,R\}), null)$
5	$((ServiceRequestor(\overline{Payment}), Table\_arrangementCart, \{R\}), Component\_CookieManager)$
6	$((ServiceRequestor(\overline{Payment}), Table\_arrangements, \{W\}), Component\_CookieManager)$
7	$((ServiceRequestor(\overline{Payment}), Table\_flowers, \{W\}), Component\_CookieManager)$
8	$((ServiceRequestor(\overline{Payment}), Table\_carts, \{W\}), Component\_CookieManager)$
9	$((ServiceRequestor(\overline{Payment}), Table\_flowerCart, \{R\}), Component\_CookieManager)$
10	$((ServiceRequestor(\overline{Payment}), Table\_sessions, \{R,W\}), Component\_CookieManager)$
11	$((ServiceRequestor(\overline{Payment}), Table\_users, \{W\}), Component\_CookieManager)$
12	$((ServiceRequestor(\overline{Payment}), Table\_arrangementCart, \{W\}), Component\_CookieManager)$
13	$((ServiceRequestor(\overline{Payment}), Table\_flowerCart, \{W\}), Component\_CookieManager)$
14	$((ServiceRequestor(\overline{Payment}), Table\_carts, \{W\}), Component\_CookieManager)$

Service PMPs provide the foundation for software architects to conduct mechanism strength analysis for the privileges involved in the service. For instance, we can capture quantitatively the mechanism strength of the privileges involved in the *Payment* service using the *PMS* Metric defined earlier. For instance, if we assume that the mechanism strength of the *Component_CookieManager* is a number *a*, then we can evaluate according to Table 5.8 the privilege mechanism strength for each of the privileges p_r in service $\overline{Payment}$ as $PMS(p_r, \overline{Payment}) = a$. Based on PMS metrics, software architects can therefore identify unprotected privileges or determine whether the protections of specific privileges should be enhanced.

5.5 Summary

In this chapter, we discuss architecture security analysis based on the USIE model. As indicated above, our approach consists of computing metrics for specific internal software attribute and using the computed metrics to assess the level of exposure to attackability. The interpretation of the internal metrics is based on the empirical security design ***hypotheses*** outlined in chapter 3. Particularly, we consider the metrics defined in this chapter to represent only sample metrics that can be derived from our framework, but we do not claim that these metrics are mature metrics and ready to be adopted by the software industry. We agree that more meaningful and significant metrics can be derived by investigating pre-established security analysis theories and models, but we at the current stage of our research focus on the development of the measurement framework. Since these ***hypotheses*** represent the foundation of our quantitative security analysis approach, we need to establish their meaningfulness through empirical investigations. We tackle this issue in the next chapter through the methodology of case studies.

Chapter 6

Empirical Studies

In this chapter, we study empirically the security design hypotheses proposed in our framework. Our studies focus on the verification of the relationships between attackability, as an external software attribute, and the internal security-related attributes introduced in this work. Our approach to attack measurement is attack-specific. In particular, in this chapter, we study two different types of software attacks, namely URL Jumping attack and denial of service attack. These allow us to study hypotheses 1 and 2 (see section 3.1.3). The methodology employed in these studies can easily be adapted and applied for hypotheses 3 and 4, in which case possible attacks such as password-cracking attack and privilege escalation attack can be investigated. We postpone such study for future work.

We start this chapter by describing the general context of our studies. Then we introduce a general attackability measurement approach and provide as instances the attackability metrics for URL Jumping attack and denial of service attack. The rest of the chapter describes the experiment environment and presents the details of the empirical study conducted using two different service-oriented software applications, namely a Flower Shop application and a MVN[2] Forum application.

6.1 Context

Empirical validation is essential to confirm or disconfirm any claim about the relationships between software external and internal attributes. However, most claims in recent software engineering papers were not rigorously validated or supported by

[2] MVN is the abbreviation for My VietNam Group that develops a variety of web applications in Vietnam.

empirical evidence. Software engineering practices in the current stage are not mature enough to establish significant empirical methods due to the massive amount of subjective factors involved in the software development processes. Also the diversity and rapid changes of software development techniques make it very difficult to obtain a statistical significant sample set for empirical research. However, we believe that empirical results obtained from restricted context or using limited resources are still valuable in the sense that they can provide supportive evidence in the specified context. What is important is that the criteria of the empirical validation be clearly defined so that the empirical study can be repeated to other contexts and resources.

According to Briand [7], empirical validations of relationships between internal and external attributes mainly involve three steps:

(1) *Data collection for both internal and external attributes metrics.*
(2) *Measurement level identification of both internal and external metrics.*
(3) *Selection and use of appropriate analysis approaches to establish the relationships.*

Step 1 is straightforward; so is step 3 once step 2 is addressed. Step 2 is hardly straightforward, especially in the discipline of software engineering. As noted by Briand [7], the determination of a scale type of a software product metric is basically intuitive. The general approach to address the scale type problem in many science disciplines is to assume different scale types and then compare the results using related statistical techniques.

In our work, we consider our current measurement scheme to be on an ordinal scale. Therefore, we simply study the relationships between internal attribute metrics and corresponding attackability metrics using ordinal level data analysis techniques.

Specifically, we selected *Pearson product-moment* correlation analysis in this dissertation to establish the relationship between internal security metrics and external software attackability. In statistics, the *Pearson product-moment correlation* analysis is a common approach to analyze the correlation relationship between two variables X and Y [81]. A *Pearson product-moment correlation* coefficient reflects the degree of linear relationship between the two variables and ranges from -1 to +1. Basically, a +1

correlation coefficient means that there is a perfect positive linear relationship between variables; a -1 correlation coefficient indicates that there is a perfect negative linear relationship between variables; and a zero correlation coefficient means there is no linear relationship between the two variables. Certain outcome of the correlation coefficient between -1 and +1 could indicate whether correlations are strong or weak. Particularly, when the variable data comes from a sample instead of a population, the statistic of the *Pearson product-moment* correlation coefficient is defined by the following formula:

$$r = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{X_i - \overline{X}}{s_X}\right)\left(\frac{Y_i - \overline{Y}}{s_Y}\right)$$

Where r represents the correlation coefficient; n represents the sample size; X_i represents the a sample data of variable X; $\overline{X}$ represents the sample mean of variable X; Y_i represents a sample data of variable Y; $\overline{Y}$ represents the sample mean of variable Y; s_X and s_Y represent respectively the sample standard deviation of variable X and Y.

In order to further test the strength of the correlation coefficient, we apply for each case study hypothesis testing and confidence interval analysis on the set of correlation coefficients collected from several independent attackability analysis sessions. Specifically, in order to apply hypothesis testing and confidence interval analysis on a sample data, we need to assume the distribution of the correlation coefficients (since the true distribution of the correlation coefficients is unknown). In statistics, the use of the normal model can be theoretically justified by assuming that many small, independent effects are additively contributing to each observation. In addition, the normal distribution maximizes information entropy among all distributions with known mean and variance, which makes it the natural choice of underlying distribution for data summarized in terms of sample mean and variance. In practise, many measurements can be approximated well by normal distribution even if the distribution of the population from which the sample is taken is not normal [81]. Generally, the normal distribution is the most widely used family of distributions in statistics and many statistical tests are based on the assumption of normality.

The primary objective of our empirical study is to provide empirical evidences for the existence of relationships between our proposed internal security metrics and external software attackability. Due to the diversity and complexity of software attacks, it is

neither realistic nor practical to interpret software attackability as a single attribute. Accordingly, we interpret software attackability with respect to specific software attacks.

6.2 Attackability Measurements

In this section, we firstly introduce our general approach for attackability measurement and then provide specific attackability metrics for two common application-level attacks, namely URL Jumping attack and denial of service attack.

6.2.1 General Approach

To derive attackability metrics, we adopt an *effort-reward* approach to assess the external attackability of software services in operational environments. We define specific metrics for attack effort and reward with respect to specific software attacks. The relative attackability of two different software services facing the same type of attack in identical operational environments can be captured by comparing the attack efforts required under the same reward, or by observing the attack rewards involved under the same effort. Generally, we compute the relative attackability by the ratio $\frac{Attack\ Reward}{Attack\ Effort}$. The idea of modeling attackability with *effort* and *reward* came originally from Brocklehurst *et al.* [9]. They proposed to evaluate operational software security by estimating the relationship between attack effort and attack reward. However, their methodology requires expertise and subjective effort to interpret the relationship between attack effort and reward. Since our attackability evaluation methodology is based on comparison rather than derivation of absolute measures, the relative attackability computation in our context can be objective as long as we are able to quantify the factors underlying *attack effort* and *attack reward*. In our study, we define different metrics to capture different types of *attack effort* and *attack reward* based on the specific nature of corresponding attack. The following sections present the instance metrics defined in our work to measure attackability regarding URL Jumping attack and denial of service attack.

6.2.2 URL Jumping Attackability Measurement

A common web-based application relies on a specific flow of functionality that developers expect or assume will faithfully be followed by users. For instance, in a typical e-commerce application, after placing an order a new customer is expected to register (by creating a secure account) before initiating the payment process. For a returning customer, the application typically will expect that after placing an order, the customer would login before paying. In either case there is a well-defined sequence of tasks that the developer would expect to be enforced. For instance, if a returning customer can jump directly from the Order page to the Payment page, without going through the Login page, this means that the system has no way to keep track of its customers. Obviously, this is not very effective either from business perspective or from security standpoint. Unfortunately the inherently stateless nature of the HTTP protocol makes such scenario possible. It is possible for a user to jump to any page just by entering the URL in a Web browser. The purpose of the URL Jumping attack is to exploit such deficiency in poorly developed Web application, by identifying in a web application a set of tasks that should be sequenced, and attempting to skip certain (required) steps by browsing over them [2].

In principle, the service execution sequence should be validated in the program logics by software developers. Unfortunately, some developers, due to careless implementation or misunderstanding of complex services dependencies, may omit or miss validations of the preconditions associated with key services executions. As a result, malicious users may exploit such opportunity by breaking out of page sequences. For URL jumping attack, we quantify attack effort and reward of the URL jumping attack by defining the following metrics:

Metric of URL Jumping Attack Effort:

$$AttackEffort_{URL\text{-}Jumping}(service_i) = The\ number\ of\ randomly\ exploited\ URLs\ related\ to\ the\ service_i.$$

Metric of URL Jumping Attack Reward

$$AttackReward_{URL\text{-}Jumping}(service_i) = \begin{cases} 1, if\ a\ URL\ Jumping\ vulnerability\ is\ found\ in\ service_i. \\ 0, \qquad otherwise. \end{cases}$$

Specifically, we define the attack effort for URL Jumping attack on a given composite service as the total number of the URLs explored randomly in finding a URL jumping vulnerability. We define the attack reward of URL jumping attack as a binary value which will be set to 1 if there is URL Jumping vulnerability in the given composite service and 0 otherwise.

Since there is no universal interpretation for the notions of attack effort and reward for URL Jumping attack pattern, we define in this work corresponding metrics based on our own understanding. In this definition, we assume that attackers perform the URL Jumping attack using a single approach that is to break the application logic by randomly accessing URLs directly. We believe that our metrics capture URL Jumping attackability to some extent, but we do not claim that these definitions are definitive. Defining universal attackability metrics is not in the scope of this work.

6.2.3 Denial of Service Attackability Measurement

Denial of Service (DOS) attacks are attempts by unauthorized users to prevent legitimate users from accessing computing resources or services. These kinds of attacks may target a network, a host, or specific application running on a host. The primary objective of application-level DOS attacks, which are of particular interest in software security engineering, is to prevent application specific services from being available to legitimate users rather than collapsing the application's hosting computer. Usually, an application DOS attack starts by traversing the application services that are publicly available and finding the most costly services in terms of response time. After locating such services, the attackers will try to degrade the performance of the application by submitting massive number of requests to those services. A successful application DOS attack will result in the poor performance of not only the targeted services, but also of some other application services. Typically, in the application DOS attack pattern, the services used by attackers to perform the attack are referred to as target services, and the services whose performances are affected are called victim services [2].

In this work, for a given target service we measure DOS attack effort by computing the ratio between the workload under attack and the workload under regular condition as

Metric of DOS Attack Effort:

$$AttackEffort_{DOS}(targetService) = \frac{AttackWorkload(targetService)}{RegularWorkload(targetService)}$$

For a given victim service under attack condition, we quantify DOS attack reward by computing the ratio between its response time and its response time under regular condition as

Metric of DOS Attack Reward:

$$AttackReward_{DOS}(victimService) = \frac{ResponseTimeUnderAttack(victimService)}{ResponseTimeUnderRegularCondition(victimService)}$$

Regular conditions are defined by the software requirements. In the above formulas, we need specifications of the expected workloads and response times for the different services. Identifying required workload and response time values is already an established practice in performance analysis and capacity planning for web services [53].

Based on the above formulas, we define DOS attackability for a given target relative to a given victim service as the ratio $\frac{AttackReward_{DOS}(victimService)}{AttackEffort_{DOS}(targetService)}$, and refer to this metric as *relative DOS attackability*.

In our definition of application DOS attackability measurement, we assume that attackers have zero knowledge on the application logic and simply perform the attack based on blind URL access. Like with our URL jumping attackability measurement, we believe that our DOS attackability metric captures the application DOS attackability to some extent, but we do not claim that these metrics are universal.

6.3 Experiment Environment

In our study, we conducted experiments using two online applications, namely the Flower Shop application and the Forum application. Both of the applications require a web server and a database server for deployment. The web server is used to host application logic components and the database server simply manages the backend data for corresponding applications. Specifically, we deployed our target applications on two different server machines: one running a DeveloperSide.Net web server version 1.16, and the other deploying a database server MySQL 5.1 serving as central data storage. Attacks were conducted remotely from a different machine.

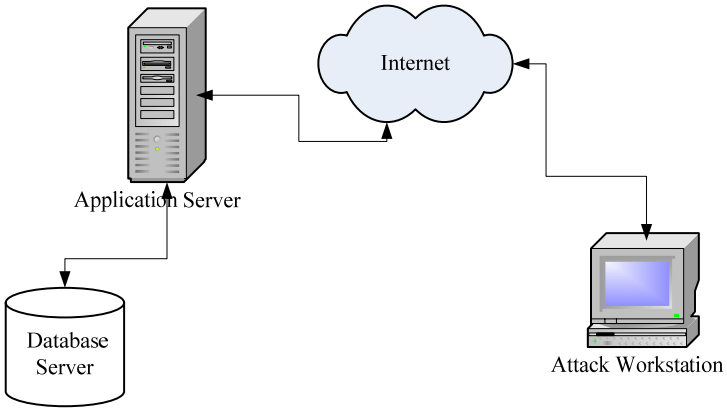

Figure 6.1 Experimental environment

Figure 6.1 illustrates the general architecture of the distributed system. The characteristics of the server machines involved are as follow:

- Application server computer: Intel ® Pentium 4 Mobile CPU, 2.0 GHz, 522MB RAM, Win 2000.
- Database server computer: Intel ® Pentium 3 CPU, 1.0 GHz, 224MB RAM, Windows XP.
- Attack workstation computer: Intel ® Pentium 4 Mobile CPU, 2.0 GHz, 522MB RAM, Win XP.

6.4 Empirical Study Using Flower Shop Application

In this study, we use as target application the FS application, which as indicated earlier is an open source web-based software system. The FS system has already been described in the previous chapters. The rest of the section presents and analyzes the measurement results regarding URL Jumping attack and Denial of Service attack.

6.4.1 Study based on URL Jumping Attack

Our objective in this study is to analyze the service architecture of the flower shop system to understand which design-level security metrics, in the application context, are likely to bear on its ability to resist to URL Jumping attack. Based on our intuitive understanding of how this attack operates, we make the assumption that *service URL Jumping attackability increases as service complexity increases*. This is consistent with hypothesis 1 given in chapter 3. We use the *ASD* metric defined in chapter 5 to compute service complexity.

6.4.1.1 Measurement Results

As shown in Figure 4.13, the FS application consists of 14 composite services. In this study, we used all these 14 composite services as the target services to validate our hypothesis empirically. We present in the following the complexity and attackability measurements obtained, and discuss the corresponding results.

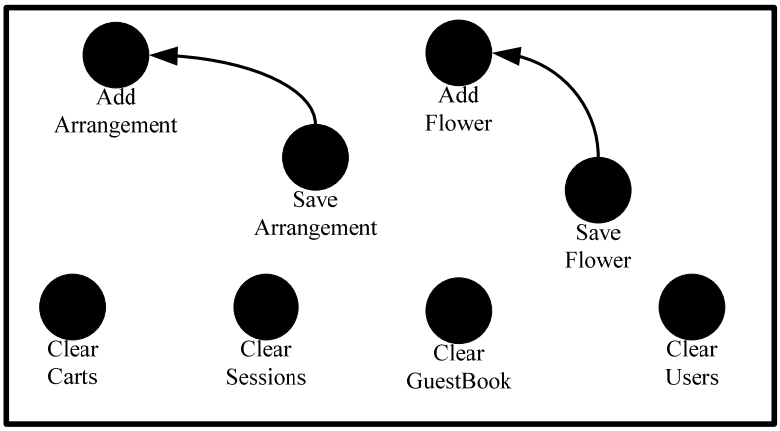

Figure 6.2. The USIE model of Administrator Service

USIE based complexity analysis starts by deriving the USIE configuration graphs of the composite services. Figures 6.2 ~ 6.14 show respectively the USIE configuration graphs constructed for the FS composite services except for the Ordering service. The USIE configuration graph of the Ordering service was shown earlier in Figure 4.11.

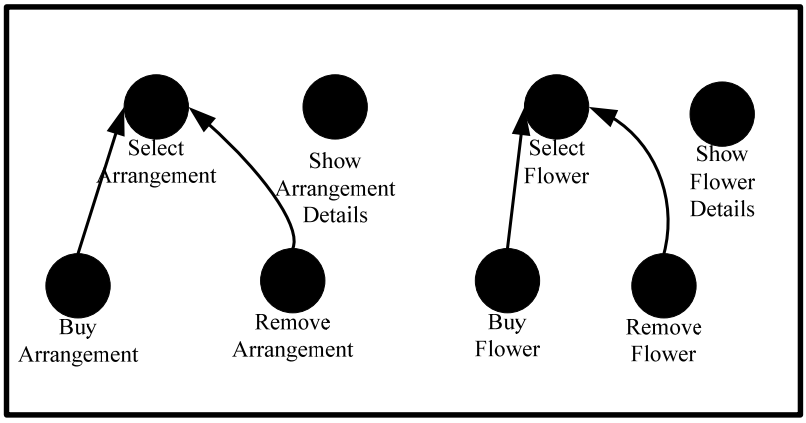

Figure 6.3. The USIE model of Customer Shopping Service

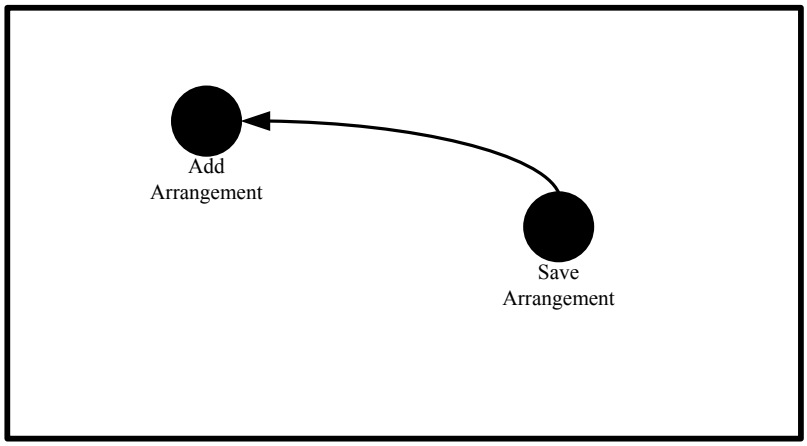

Figure 6.4. The USIE Model of Arrangement Management Service

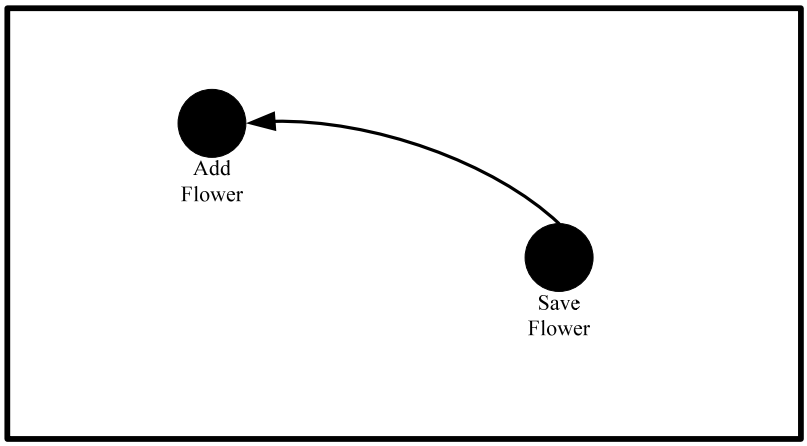

Figure 6.5. The USIE Model of Flower Management Service

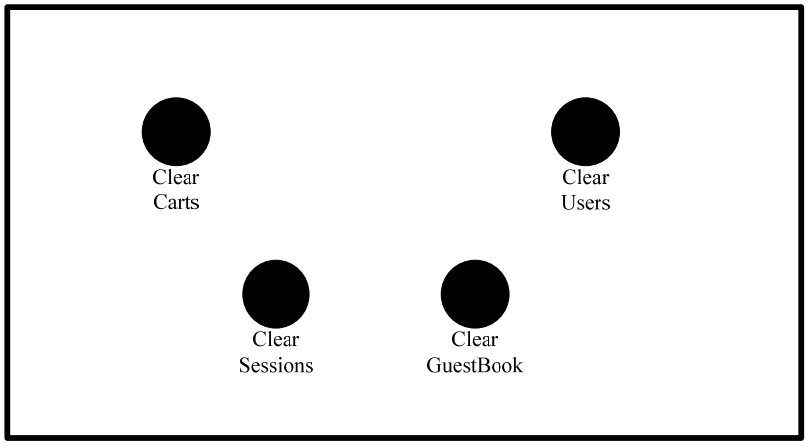

Figure 6.6. The USIE Model of User Management Service

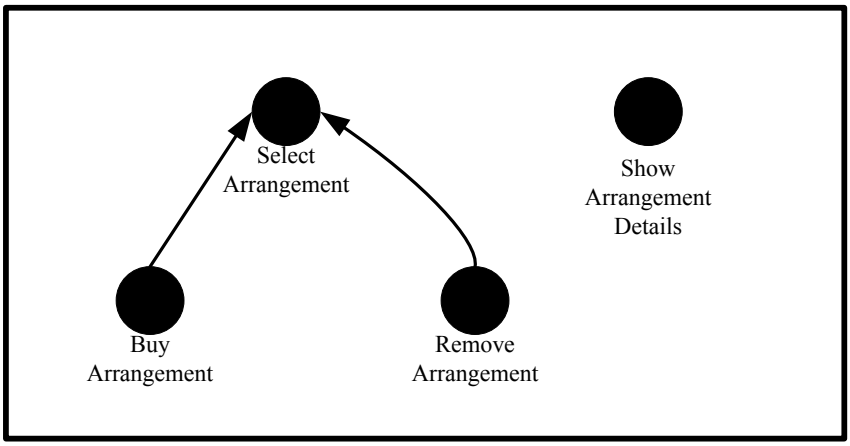

Figure 6.7. The USIE Model of Arrangement Service

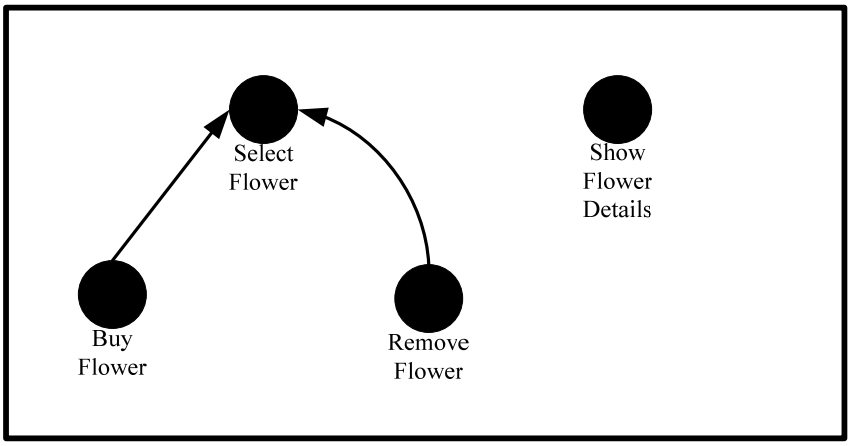

Figure 6.8. The USIE Model of Flower Service

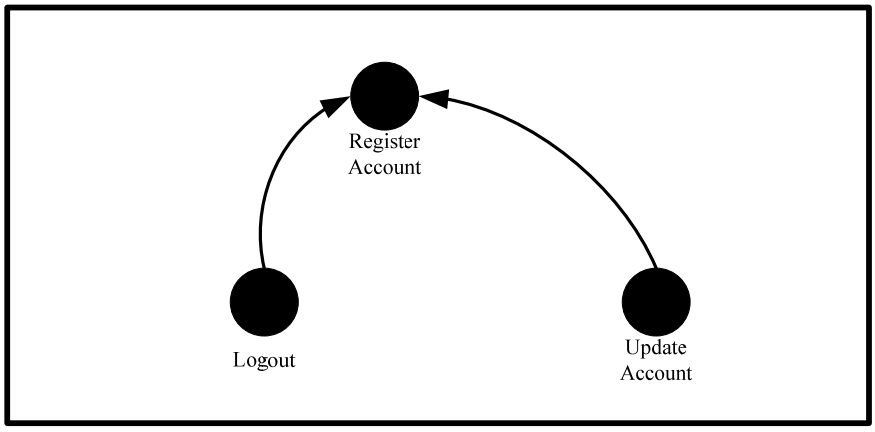

Figure 6.9. The USIE Model of Account Service for New Users

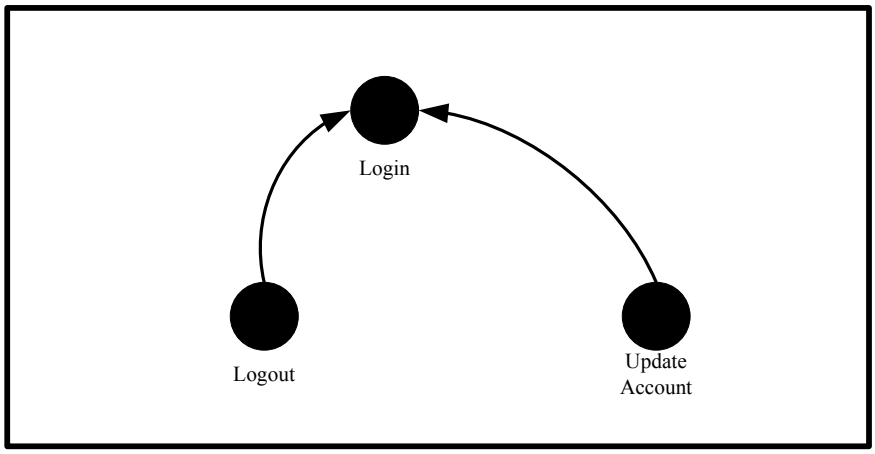

Figure 6.10. The USIE Model of Account Service for Existing Users

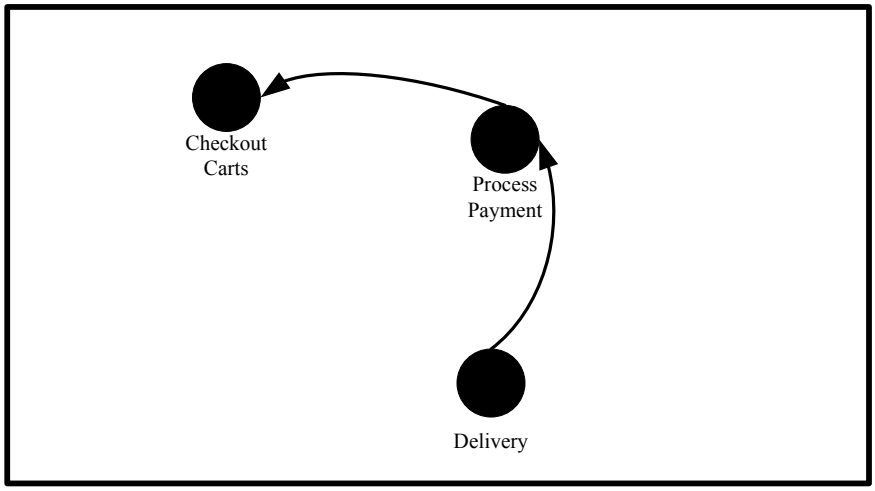

Figure 6.11. The USIE Model of Payment Service

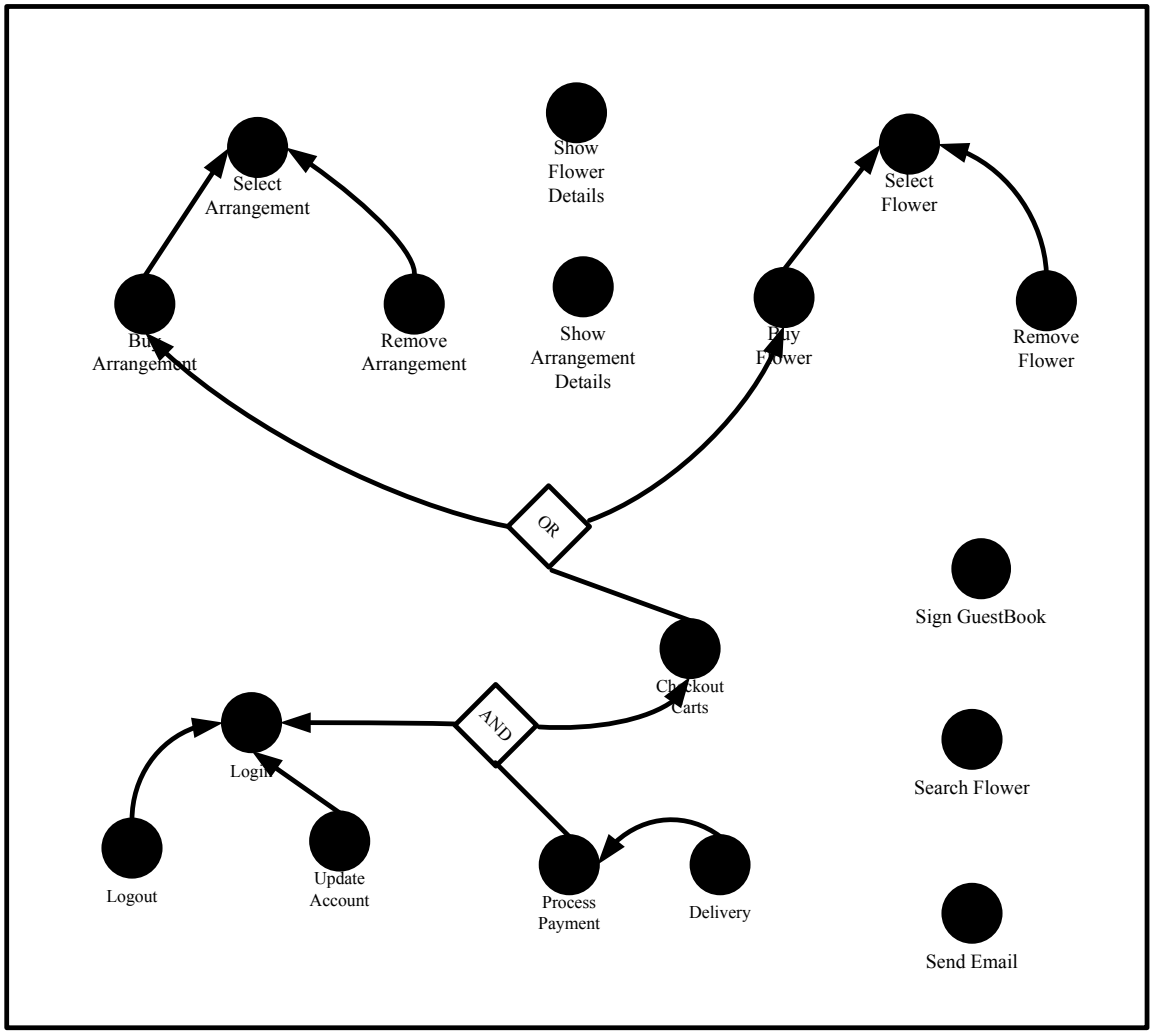

Figure 6.12. The USIE Model of Customer Service

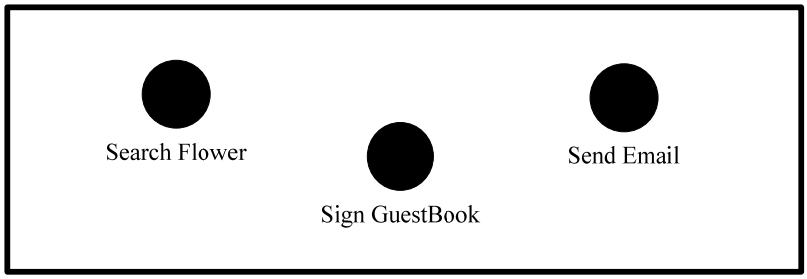

Figure 6.13. The USIE Model of Miscellaneous Service

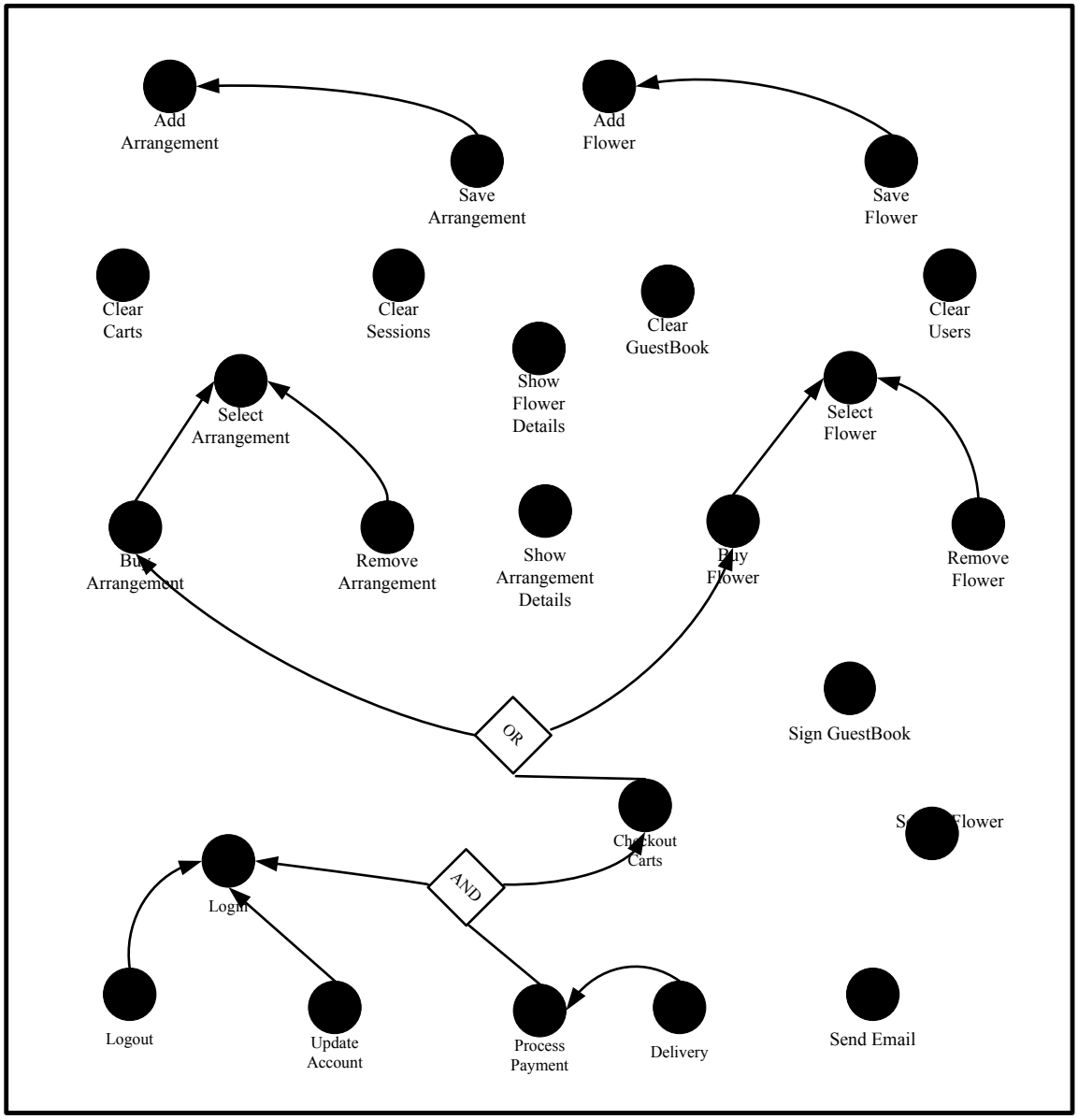

Figure 6.14. The USIE Model of Flower Shop Service

Based on the configuration graphs of FS composite services, we derive the use patterns for each of the composite service using the algorithm introduced earlier in Table 5.1. Then, we compute complexity values based on the metric definition introduced in section 5.1.3. Table 6.1 shows the corresponding *ASD* values for each of the 14 composite services involved in the FS application. Note that the *ASD* values of some composite services are higher than other composite services, which indicates that these services are more complex than others in term of *ASD* metric. Therefore, we can intuitively conclude that the attackability underlying the service designs with higher *ASD* values are relatively

greater compared with other services with lower *ASD* values. In fact, we confirmed later in our analysis that the services with higher *ASD* values are more vulnerable to URL Jumping attack than the services with lower *ASD* values.

Table 6.1 ASD Metric Values

No.	***Composite Service*** $\bar{s}$	***ASD(*** $\bar{s}$ ***)***
1	Payment Service	1
2	Account Service for Existing Users	0.67
3	Account Service for New Users	0.67
4	Flower Service	0.5
5	Arrangement Service	0.5
6	Flower Management Service	0.5
7	Arrangement Management Service	0.5
8	User Management Service	0
9	Miscellaneous Service	0
10	Flower Shop Service	1.08
11	Customer Service	1.47
12	Administrator Service	0.25
13	Shopping Service	0.5
14	Ordering Service	2

Generally, the measurement result is significant for both software architects and developers in the sense that software architects can pay more attention to the service design and minimize security at the architecture level, and software developers can use the measurement result as a guide to prioritize their efforts regarding security development.

For attackability measurement, we conduct the URL Jumping experiments over 20 different runs. In each independent run of the experiments, we randomly choose URLs from the identified URL sets of each target services to look for URL Jumping vulnerability. If no URL Jumping vulnerability is found for a target composite service after attacking all of its atomic services, we consider the corresponding attack effort to be infinite. Specifically, we decided, for the URL Jumping attack, to compare relative service attackability under the same attack reward by posing *AttackReward = 1*. Therefore, in each run of the experiments, the attacks on the corresponding composite services will stop once we are able to identify a violation of the application logic when exploiting URLs for the corresponding composite services.

Table 6.2 (a) URL Jumping Attack Effort under *AttackReward = 1*

	Experiment No.									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Payment Service	1	2	2	2	1	2	2	1	2	2
Account Service for Existing Users	3	1	2	3	2	3	1	3	3	3
Account Service For New Users	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Flow Service	3	4	2	3	4	3	3	2	1	4
Arrangement Service	4	4	3	3	2	3	1	4	3	3
Flow Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Arrangement Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
User Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Miscellaneous Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Flow Shop Service	10	14	7	6	11	9	13	6	7	10
Customer Service	5	5	2	6	3	5	5	3	4	3
Administrator Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Shopping Service	6	6	3	7	4	5	6	4	6	4
Ordering Service	1	1	2	1	1	1	2	1	1	2

Accordingly, the relative attackability between these services can be compared based on their attack efforts. Table 6.2 (a) presents the results of the URL Jumping attacks over 10 independent runs numbered 1~10, and Table 6.2 (b) presents the results of the URL Jumping attacks over the other 10 independent runs numbered 11~20,. Table 6.3 (a) and

(b), which are derived from Table 6.2 (a) and (b), present the relative URL Jumping attackability for the corresponding fourteen composite services in each of the runs.

Table 6.2 (b) URL Jumping Attack Effort under *AttackReward = 1* (continued)

	Experiment No.									
	11	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
Payment Service	1	2	2	1	2	1	2	2	2	1
Account Service for Existing Users	2	2	3	2	3	1	3	1	1	3
Account Service For New Users	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Flow Service	1	3	4	3	2	4	3	4	3	3
Arrangement Service	3	3	2	4	3	4	4	4	3	4
Flow Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Arrangement Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
User Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Miscellaneous Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Flow Shop Service	6	14	11	12	17	7	9	11	8	13
Customer Service	5	3	3	5	4	4	4	7	3	6
Administrator Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Shopping Service	5	4	4	7	7	7	5	7	5	7
Ordering Service	1	2	1	1	1	1	1	2	2	1

Table 6.3 (a) Relative URL Jumping Attackability

	Experiment No.									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Payment Service	1.00	0.50	0.50	0.50	1.00	0.50	0.50	1.00	0.50	0.50
Account Service for Existing Users	0.33	1.00	0.50	0.33	0.50	0.33	1.00	0.33	0.33	0.33
Account Service For New Users	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Flow Service	0.33	0.25	0.50	0.33	0.25	0.33	0.33	0.50	1.00	0.25
Arrangement Service	0.25	0.25	0.33	0.33	0.50	0.33	1.00	0.25	0.33	0.33
Flow Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Arrangement Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
User Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Miscellaneous Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Flow Shop Service	0.10	0.07	0.14	0.17	0.09	0.11	0.08	0.17	0.14	0.10
Customer Service	0.20	0.20	0.50	0.17	0.33	0.20	0.20	0.33	0.25	0.33
Administrator Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Shopping Service	0.17	0.17	0.33	0.14	0.25	0.20	0.17	0.25	0.17	0.25
Ordering Service	1.00	1.00	0.50	1.00	1.00	1.00	0.50	1.00	1.00	0.50

Table 6.3 (b) Relative URL Jumping Attackability (continued)

	Experiment No.									
	11	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
Payment Service	1.00	0.50	0.50	1.00	0.50	1.00	0.50	0.50	0.50	1.00
Account Service for Existing Users	0.50	0.50	0.33	0.50	0.33	1.00	0.33	1.00	1.00	0.33
Account Service For New Users	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Flow Service	1.00	0.33	0.25	0.33	0.50	0.25	0.33	0.25	0.33	0.33
Arrangement Service	0.33	0.33	0.50	0.25	0.33	0.25	0.25	0.25	0.33	0.25
Flow Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Arrangement Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
User Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Miscellaneous Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Flow Shop Service	0.17	0.07	0.09	0.08	0.06	0.14	0.11	0.09	0.13	0.08
Customer Service	0.20	0.33	0.33	0.20	0.25	0.25	0.25	0.14	0.33	0.17
Administrator Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Shopping Service	0.20	0.25	0.25	0.14	0.14	0.14	0.20	0.14	0.20	0.14
Ordering Service	1.00	0.50	1.00	1.00	1.00	1.00	1.00	0.50	0.50	1.00

6.4.1.2 Analysis of the Results

As shown in the plot diagram of Figure 6.15, there seems to exist certain correlation between the *ASD* Metric and relative URL Jumping Attackability. It is worthy of digging

up further by carrying out relevant statistical analyses to test the existence of the relationship.

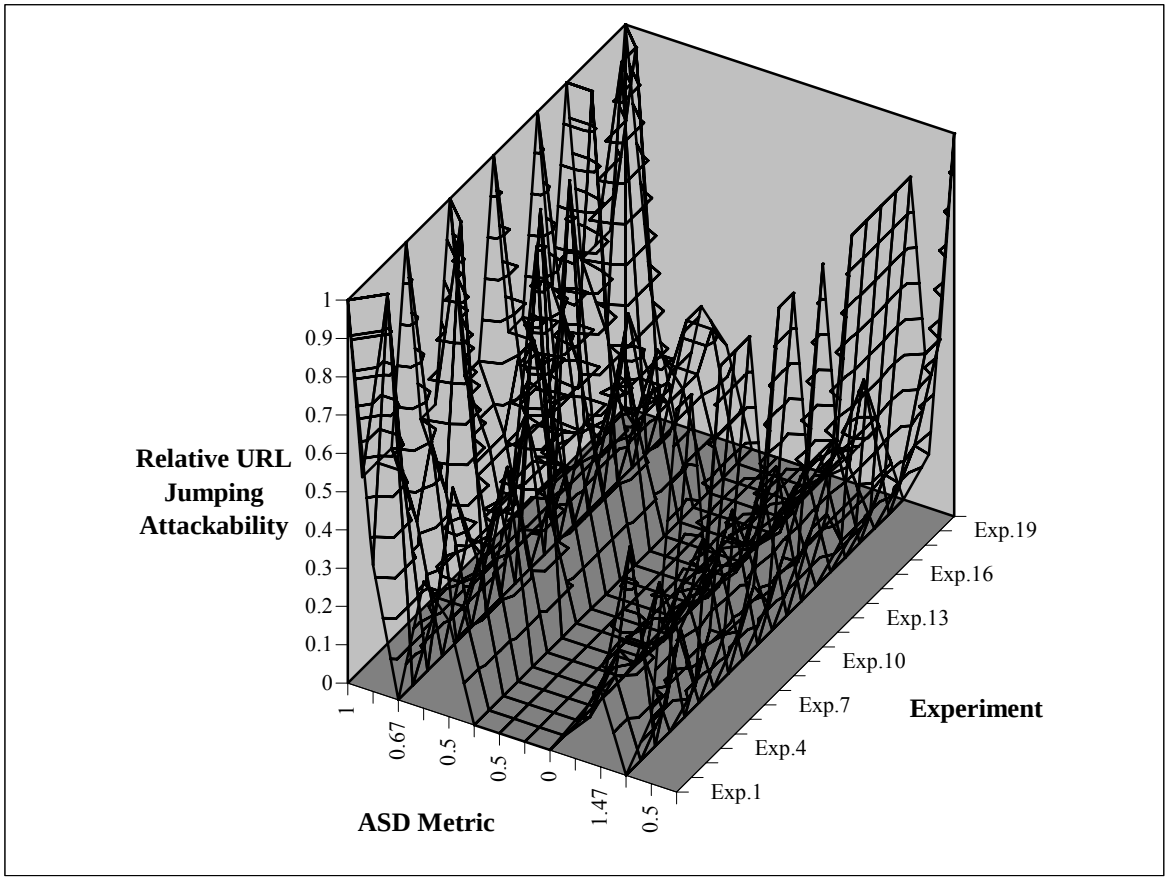

Figure 6.15. Plot Diagram for URL Jumping

At this stage, we only consider the measurement defined in this work to be ordinal scales, accordingly, we choose correlation analysis as the major analysis approach for the data collected in our experiments.

Table 6.4 presents the correlation coefficients between service ASD metrics (presented in Table 6.1) and the service relative attackability metrics (presented in Table 6.3 (a) and (b)).

Table 6.4 Correlation Coefficients between *ASD* and Relative URL Jumping Attackability

Experiment No.	***Correlation Coefficients***
1	0.681050
2	0.602200
3	0.618544
4	0.755385
5	0.678461
6	0.749005
7	0.273881
8	0.707399
9	0.573534
10	0.689459
11	0.542840
12	0.608890
13	0.757101
14	0.664145
15	0.718636
16	0.610712
17	0.780315
18	0.397275
19	0.454348
20	0.667551

Figure 6.16 presents the results of the analysis of the correlation coefficients. According to Figure 6.16, the mean of the correlation coefficients would be an estimator for the relationship between *ASD* metric and relative URL Jumping attackability. The estimate is around 0.63, which indicates a positive correlation between *ASD* metric and relative URL Jumping attackability. Hence, the relative URL Jumping attackability value increases as the *ASD* metric increases. The median of the correlation coefficients could also be a good indicator. The estimated value is around 0.67, which signals an even stronger correlation between the *ASD* metric and relative URL Jumping attackability. The histogram (see Figure 6.16) shows the relative frequencies of all the correlation coefficients from our sample. Clearly, we observe that most of the correlation coefficients fall in the range 60-80% for this sample.

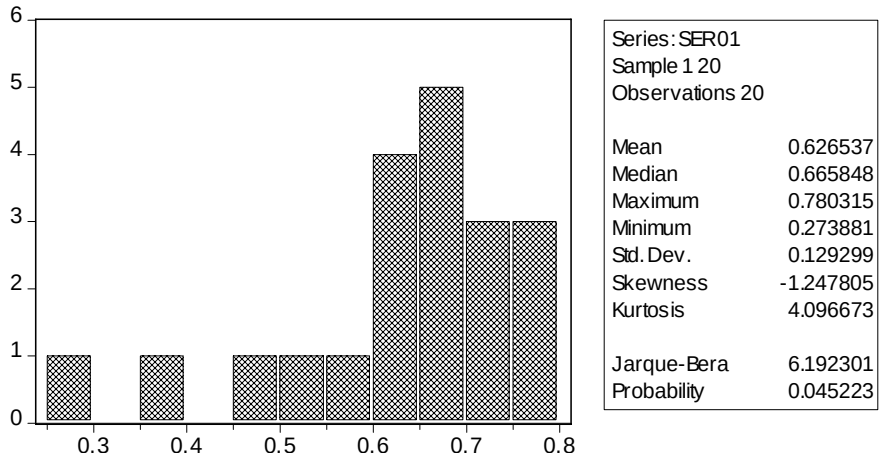

Figure 6.16. Analysis of Correlation Coefficients for URL Jumping

In order to further test the strength of the correlation between *ASD* Metric and relative URL Jumping attackability, we apply hypothesis testing and confidence interval analysis.

We begin with testing that the hypothesis of the correlation between the *ASD* Metric and relative URL Jumping attackability is always greater or equal to 0.60 on average, which is considered to be strong correlation. We establish that the null hypothesis is a composite hypothesis and that the alternative hypothesis is one-sided:

$$H_0 : Average\ Correlation \geq 0.60 \quad versus \quad H_a : Average\ Correlation < 0.60$$

We choose the confidence level to be 0.95, and assume the correlation follows a normal distribution. The appropriate test statistic has a t-distribution. With a sample size of 20, the critical value is -1.73.

Analysis of the collected correlations between *ASD* metric and relative URL Jumping attackability (See Figure 6.16) results in the mean to be 0.63, and the standard deviation to be 0.13, which yields the computed test value to be 1.03. Since the calculated value lies in the acceptance region (1.03>-1.73), we conclude that the average correlation between *ASD* metric and relative URL Jumping attackability is strong.

Another way of determining whether the correlation between *ASD* Metric and relative URL Jumping attackability is strong is to apply confidence interval analysis. The result of the analysis shows that 95% of time, the true value of the average correlation between the two variables will be covered by the confidence interval between 0.57 and 0.69 (0.57<correlation<0.69) based on the sample of the data, and therefore we conclude that the correlation is strong.

In conclusion, the relative URL Jumping attackability of a software service and the service's *ASD* metric value tend to increase in the same directions. Therefore, these results suggest that, at least under the current experimental conditions our hypothesis is supported.

6.4.2 Study based on Application DOS Attack

Our objective in this study is to analyze the service architecture of the flower shop system to understand which design-level security metrics, in the application context, are likely to bear on its ability to resist to application DOS attack. Based on our intuitive understanding of how this attack operates, we make the study hypothesis that *application DOS attackability increases as the service coupling between target services and victim services increases*. This stems from hypothesis *2* given in chapter 3. We use the *RSR* metric defined in Chapter 5 as the service coupling metric in this study.

6.4.2.1 Measurement Results

In this study, we selected several atomic services from the flower shop application as the context to evaluate empirically our hypothesis. Figure 6.17 ~ 6.24 illustrate the USIE models for the atomic services involved in this study. Specifically, 8 atomic services are chosen including *Register Account, Login, Logout, Update Account, Delivery, Checkout Cart*, *Process Payment* and *Buy Flower*. Each of the atomic service involves multiple operations. For instance, the *Register Account* atomic service allows a customer to register and set up a personal account. The service execution starts with the customer providing personal information, and the validation of the provided information. If the customer's information already exists in the system, the service execution terminates. Otherwise, four operations, as shown in the corresponding USIE graph, will be executed

sequentially to update system user account table, session table, log table and cookie manager component.

To study basic resource sharing characteristics, we need to compute the *RSR* metric on pairs of atomic services involved in this study. Due to the large number of possible pairs, we focused our study on a limited subset of them. Specifically, we selected randomly 18 pairs of atomic services from the atomic services involved in this study, each pair consisting of a target service and a possible victim service. Based on Figures 6.17 ~ 6.24, we computed coupling values using the *RSR* metrics for each of the atomic services pairs. Table 6.5 lists the corresponding measurement results.

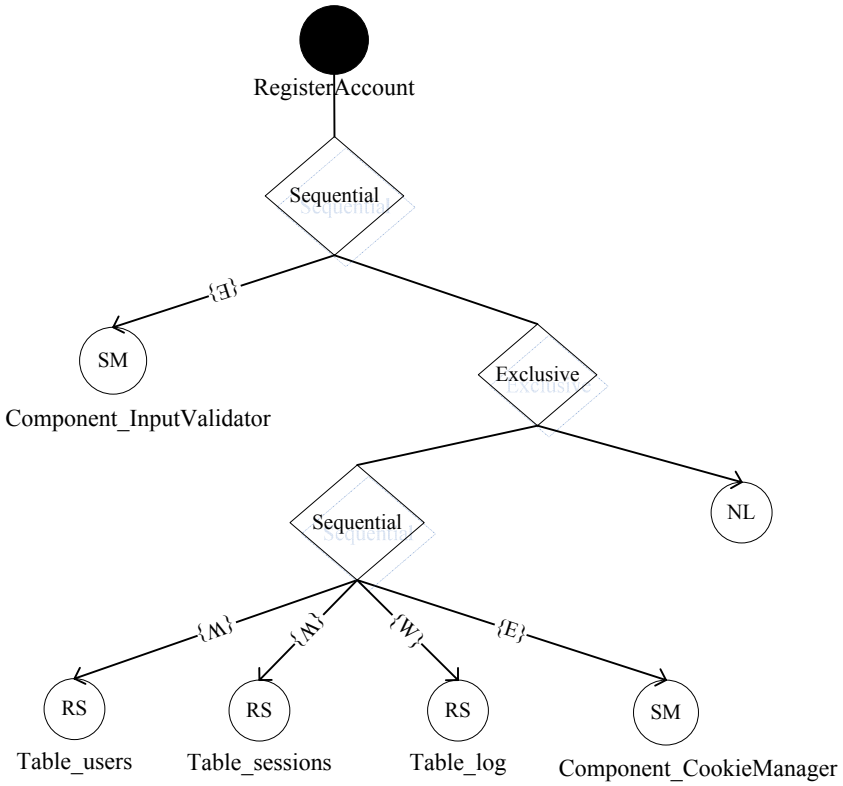

Figure 6.17. USIE Graph for the Atomic Services *RegisterAccount*

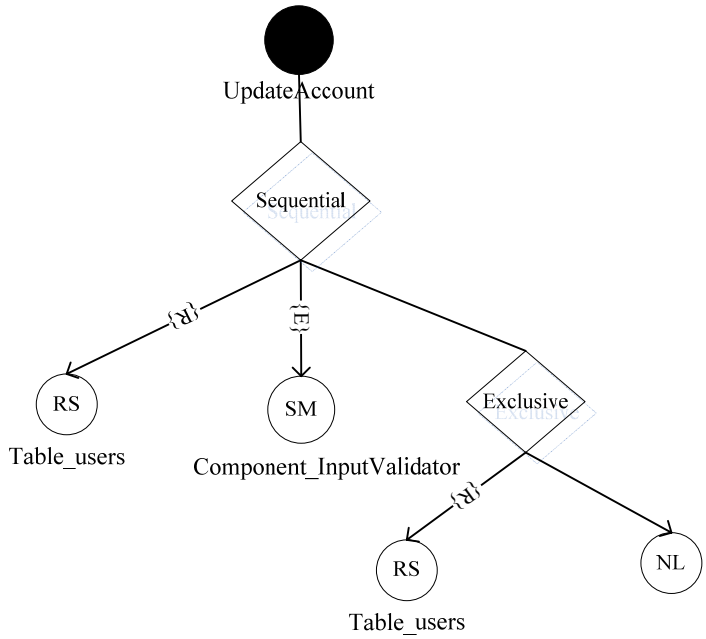

Figure 6.18. USIE Graph for the Atomic Services *UpdateAccount*

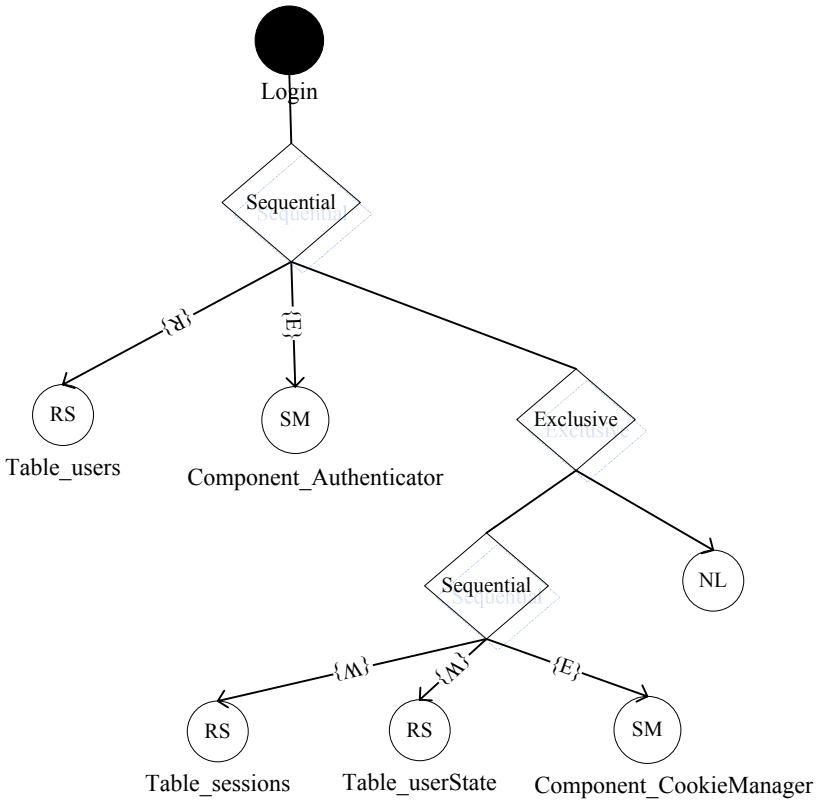

Figure 6.19. USIE Graph for the Atomic Services *Login*

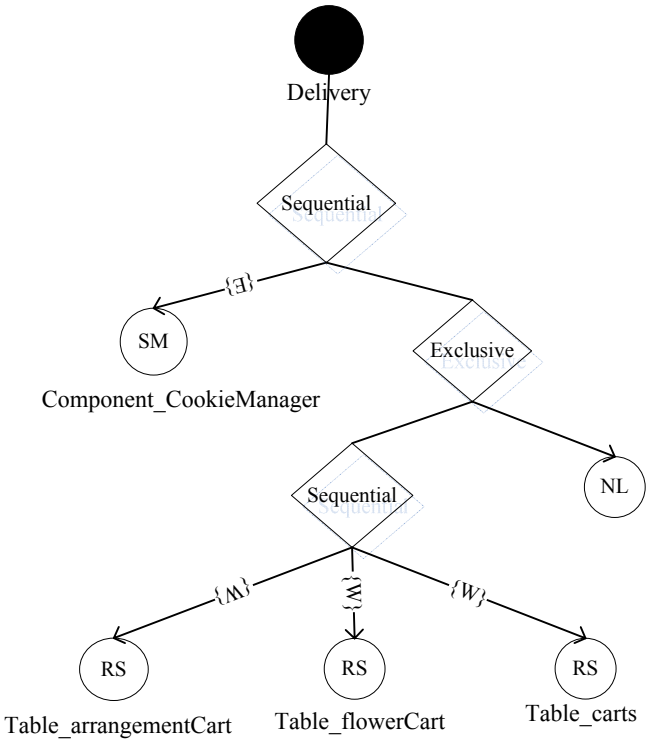

Figure 6.20. USIE Graph for the Atomic Services *Delivery*

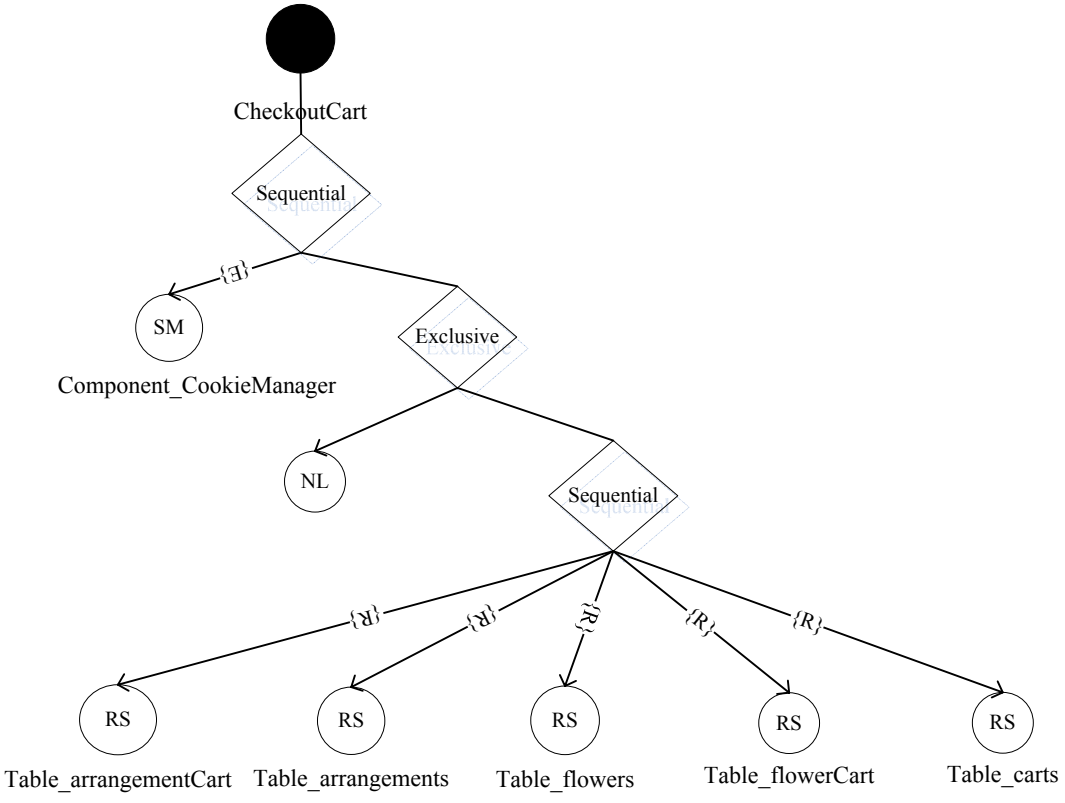

Figure 6.21. USIE Graph for the Atomic Services *CheckoutCart*

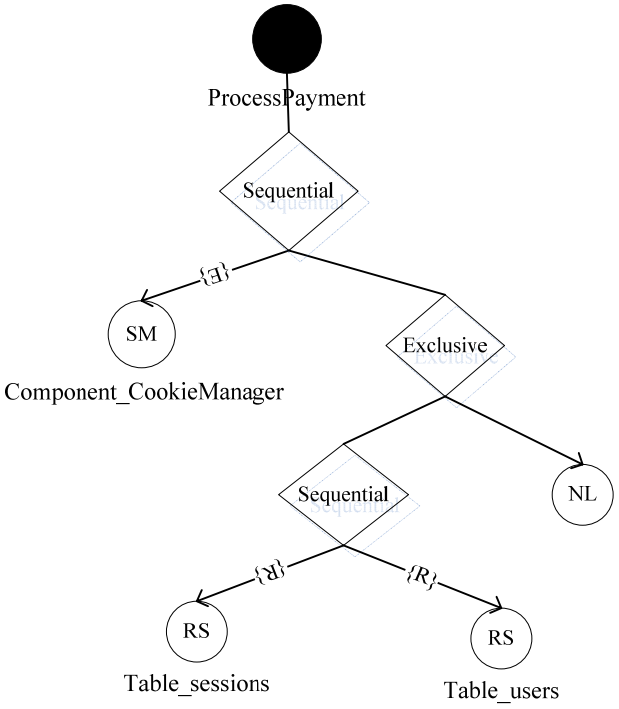

Figure 6.22. USIE Graph for the Atomic Services *ProcessPayment*

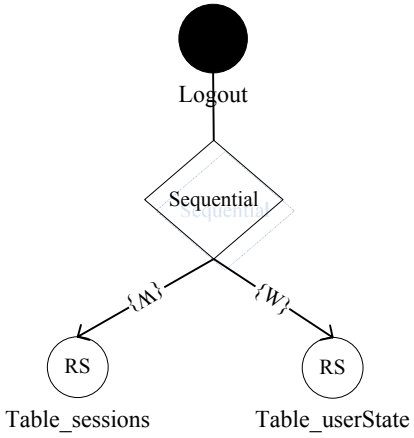

Figure 6.23. USIE Graph for the Atomic Services *Logout*

In our experiments, we used a single machine as the attack machine, which simply runs an application DOS attack script. The attack script can be configured to send varying workload to specific FS application service. The attack scripts can also measure the corresponding response time of the victim services.

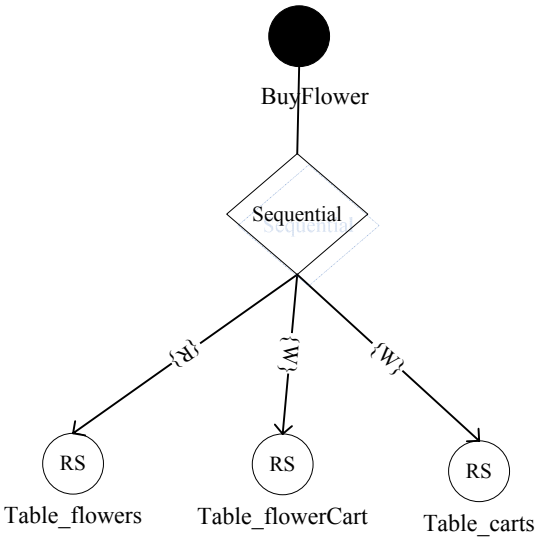

Figure 6.24. USIE Graph for the Atomic Services *BuyFlower*

The regular workload used in our attackability measurements is set to 1 *request/second*. To capture the regular response time corresponding to the regular workload for each of the atomic services considered in this study, we simply run the attack scripts under regular workload for each of the atomic services and observe the corresponding response time individually. Table 6.6 presents the regular response time observed for the atomic services involved in this study.

Table 6.5 RSR Metrics Values

No.	*Target Service*	*Victim Service*	$RSR(U_{targetService}, U_{victimService})$
1	RegisterAccount	UpdateAccount	0.33
2	Login	RegisterAccount	0.5
3	Logout	RegisterAccount	0.25
4	UpdateAccount	CheckoutCart	0
5	ProcessPayment	ProcessPayment	1
6	Delivery	UpdateAccount	0
7	CheckoutCart	Delivery	0.6
8	ProcessPayment	UpdateAccount	0.5
9	Logout	ProcessPayment	0.33
10	RegisterAccount	RegisterAccount	1
11	Login	ProcessPayment	0.67
12	RegisterAccount	ProcessPayment	0.67
13	CheckoutCart	BuyFlower	0.6
14	BuyFlower	Delivery	0.5
15	Login	Logout	0.67
16	Logout	UpdateAccount	0
17	Login	UpdateAccount	0.33
18	UpdateAccount	UpdateAccount	1

In our study, we conducted 24 rounds of DOS attacks by increasing the attack workload in each round. In each round, we submitted individually each of the target atomic services selected in our study to the same attack workload, and then computed the DOS attack reward for the corresponding victim service. As a result, we computed in each round the relative DOS attackability corresponding to the selected atomic service pairs. Table 6.7 (a) and (b) summarize the measurement results. In the table, each of the rows corresponds to an attack session or round. The second column list the attack efforts used in each round, and each of the remaining columns lists the corresponding attack rewards for 18 atomic service pairs. The numbers (see second row) used to label the service pairs match the ones used in Table 6.5.

Based on the measurements listed in Table 6.7 (a) and (b), we compute the relative DOS attackability metrics for each of the 18 pairs of atomic services in each of the 24 rounds. Table 6.8 (a) and (b) list the corresponding results.

Table 6.6 Regular Response Times of Atomic Services: Workload = 1 Request/Second

No.	***Atomic Service***	***Regular Response Time (second)***
1	RegisterAccount	0.18
2	Login	0.11
3	Logout	0.09
4	UpdateAccount	0.12
5	ProcessPayment	0.11
6	Delivery	0.10
7	CheckoutCart	0.07
8	BuyFlower	0.14

Table 6.7 (a) Measurements for DOS Attack Experiments: The columns numbered from 1 to 9 list the attack rewards corresponding to the atomic service pairs numbered from 1 to 9 in Table 6.5.

Attack Session No.	***Attack Effort***	***Attack Reward***								
		1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
1	2	8.44	9.60	8.42	3.08	9.56	3.12	9.56	6.26	9.62
2	4	7.64	9.28	7.60	3.16	9.20	3.24	9.20	6.28	9.32
3	6	8.16	10.50	8.10	3.18	10.38	3.30	10.38	6.24	10.56
4	8	7.28	10.88	7.20	3.20	10.72	3.36	10.72	6.16	10.96
5	10	9.70	9.70	6.80	3.20	11.10	3.40	10.90	6.20	9.80
6	12	6.36	11.88	6.24	3.36	11.64	3.60	11.64	6.36	12.00
7	14	5.60	12.32	7.56	3.08	12.04	3.78	11.90	6.58	12.46
8	16	7.20	12.48	7.04	3.68	12.16	4.00	12.16	6.72	12.64
9	18	6.84	12.96	6.66	3.78	12.60	4.14	12.60	7.02	13.14
10	20	6.20	13.20	6.00	3.80	12.80	4.20	12.80	7.00	13.40
11	22	6.82	13.86	6.60	3.96	13.42	4.40	13.42	7.26	14.08
12	24	6.24	14.40	6.00	4.08	13.92	4.56	13.92	7.44	14.64
13	26	5.98	15.08	5.72	4.16	14.56	4.68	14.56	7.54	15.34
14	28	5.32	15.40	5.04	4.20	14.84	4.76	14.84	7.56	15.68
15	30	4.80	16.20	4.50	4.20	15.60	4.80	15.60	7.80	16.50
16	32	5.44	16.64	5.12	4.48	16.00	5.12	16.00	7.68	16.96
17	34	5.78	18.02	5.44	4.76	17.34	5.44	17.34	8.16	18.36
18	36	5.76	18.72	5.40	5.04	18.00	5.76	18.00	8.64	19.08
29	38	5.70	19.38	5.32	5.32	18.62	6.08	18.62	8.74	19.76
20	40	6.00	20.40	5.60	5.60	19.60	6.40	19.60	9.20	20.80
21	42	5.46	21.00	5.04	5.46	20.16	6.30	20.16	9.66	21.42
22	44	6.16	21.56	5.72	5.72	20.68	6.60	20.68	10.12	22.00
23	46	5.98	22.54	5.52	5.98	21.62	6.90	21.62	10.12	23.00
24	50	7.00	24.00	6.50	6.50	23.00	7.50	23.00	11.00	24.50

Table 6.7 (b) Measurements for DOS Attack Experiments (continued): The columns numbered from 10 to 18 list the attack rewards corresponding to the atomic service pairs numbered from 10 to 18 in Table 6.5.

Attack Session No.	***Attack Effort***	***Attack Reward***								
		10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
1	2	9.54	9.48	9.38	6.38	9.64	9.66	3.15	8.54	9.62
2	4	9.16	9.04	8.84	6.52	9.36	9.40	3.29	7.84	9.32
3	6	10.32	10.14	9.84	6.60	10.62	10.68	3.38	8.46	10.56
4	8	10.64	10.40	10.00	6.64	11.04	11.12	3.46	7.68	10.96
5	10	9.40	10.50	10.00	6.80	9.90	10.00	3.53	7.40	9.80
6	12	11.52	11.16	10.56	7.08	12.12	12.24	3.76	6.96	12.00
7	14	11.90	11.34	10.64	7.42	12.60	12.74	3.54	8.40	12.46
8	16	12.00	11.52	10.72	7.68	12.80	12.96	4.21	8.00	12.64
9	18	12.42	11.88	10.98	8.10	13.32	13.50	4.37	7.74	13.14
10	20	12.60	12.00	11.00	8.20	13.60	13.80	4.46	7.20	13.40
11	22	13.20	12.54	11.44	8.58	14.30	14.52	4.69	7.92	14.08
12	24	13.68	12.96	11.76	8.88	14.88	15.12	4.87	7.44	14.64
13	26	14.30	13.52	12.22	9.10	15.60	15.86	5.02	7.28	15.34
14	28	14.56	13.72	12.32	9.24	15.96	16.24	5.12	6.72	15.68
15	30	15.30	14.40	12.90	9.60	16.80	17.10	5.19	6.30	16.50
16	32	15.68	14.72	13.12	9.60	17.28	17.60	5.54	7.04	16.96
17	34	17.00	15.98	14.28	10.20	18.70	19.04	5.88	7.48	18.36
18	36	17.64	16.56	14.76	10.80	19.44	19.80	6.23	7.56	19.08
29	38	18.24	17.10	15.20	11.02	20.14	20.52	6.57	7.60	19.76
20	40	19.20	18.00	16.00	11.60	21.20	21.60	6.92	8.00	20.80
21	42	19.74	18.48	16.38	12.18	21.84	22.26	6.85	7.56	21.42
22	44	20.24	18.92	16.72	12.76	22.44	22.88	7.17	8.36	22.00
23	46	21.16	19.78	17.48	12.88	23.46	23.92	7.50	8.28	23.00
24	50	22.50	21.00	18.50	14.00	25.00	25.50	8.15	9.50	24.50

Table 6.8 (a) Relative DOS Attackability Values, derived from Table 6.7 (a), for each of the selected atomic service pairs.

Attack Session No.	***Relative DOS Attackability***								
	1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
1	4.22	4.80	4.21	1.54	4.78	1.56	4.78	3.13	4.81
2	1.91	2.32	1.90	0.79	2.30	0.81	2.30	1.57	2.33
3	1.36	1.75	1.35	0.53	1.73	0.55	1.73	1.04	1.76
4	0.91	1.36	0.90	0.40	1.34	0.42	1.34	0.77	1.37
5	0.97	0.97	0.68	0.32	1.11	0.34	1.09	0.62	0.98
6	0.53	0.99	0.52	0.28	0.97	0.30	0.97	0.53	1.00
7	0.40	0.88	0.54	0.22	0.86	0.27	0.85	0.47	0.89
8	0.45	0.78	0.44	0.23	0.76	0.25	0.76	0.42	0.79
9	0.38	0.72	0.37	0.21	0.70	0.23	0.70	0.39	0.73
10	0.31	0.66	0.30	0.19	0.64	0.21	0.64	0.35	0.67
11	0.31	0.63	0.30	0.18	0.61	0.20	0.61	0.33	0.64
12	0.26	0.60	0.25	0.17	0.58	0.19	0.58	0.31	0.61
13	0.23	0.58	0.22	0.16	0.56	0.18	0.56	0.29	0.59
14	0.19	0.55	0.18	0.15	0.53	0.17	0.53	0.27	0.56
15	0.16	0.54	0.15	0.14	0.52	0.16	0.52	0.26	0.55
16	0.17	0.52	0.16	0.14	0.50	0.16	0.50	0.24	0.53
17	0.17	0.53	0.16	0.14	0.51	0.16	0.51	0.24	0.54
18	0.16	0.52	0.15	0.14	0.50	0.16	0.50	0.24	0.53
29	0.15	0.51	0.14	0.14	0.49	0.16	0.49	0.23	0.52
20	0.15	0.51	0.14	0.14	0.49	0.16	0.49	0.23	0.52
21	0.13	0.50	0.12	0.13	0.48	0.15	0.48	0.23	0.51
22	0.14	0.49	0.13	0.13	0.47	0.15	0.47	0.23	0.50
23	0.13	0.49	0.12	0.13	0.47	0.15	0.47	0.22	0.50
24	0.14	0.48	0.13	0.13	0.46	0.15	0.46	0.22	0.49

Table 6.8 (b) Relative DOS Attackability Values (continued), derived from Table 6.7 (b), for each of the selected atomic service pairs.

Attack Session No.	***Relative DOS Attackability***								
	10	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
1	4.77	4.74	4.69	3.19	4.82	4.83	1.57	4.27	4.81
2	2.29	2.26	2.21	1.63	2.34	2.35	0.82	1.96	2.33
3	1.72	1.69	1.64	1.10	1.77	1.78	0.56	1.41	1.76
4	1.33	1.30	1.25	0.83	1.38	1.39	0.43	0.96	1.37
5	0.94	1.05	1.00	0.68	0.99	1.00	0.35	0.74	0.98
6	0.96	0.93	0.88	0.59	1.01	1.02	0.31	0.58	1.00
7	0.85	0.81	0.76	0.53	0.90	0.91	0.25	0.60	0.89
8	0.75	0.72	0.67	0.48	0.80	0.81	0.26	0.50	0.79
9	0.69	0.66	0.61	0.45	0.74	0.75	0.24	0.43	0.73
10	0.63	0.60	0.55	0.41	0.68	0.69	0.22	0.36	0.67
11	0.60	0.57	0.52	0.39	0.65	0.66	0.21	0.36	0.64
12	0.57	0.54	0.49	0.37	0.62	0.63	0.20	0.31	0.61
13	0.55	0.52	0.47	0.35	0.60	0.61	0.19	0.28	0.59
14	0.52	0.49	0.44	0.33	0.57	0.58	0.18	0.24	0.56
15	0.51	0.48	0.43	0.32	0.56	0.57	0.17	0.21	0.55
16	0.49	0.46	0.41	0.30	0.54	0.55	0.17	0.22	0.53
17	0.50	0.47	0.42	0.30	0.55	0.56	0.17	0.22	0.54
18	0.49	0.46	0.41	0.30	0.54	0.55	0.17	0.21	0.53
29	0.48	0.45	0.40	0.29	0.53	0.54	0.17	0.20	0.52
20	0.48	0.45	0.40	0.29	0.53	0.54	0.17	0.20	0.52
21	0.47	0.44	0.39	0.29	0.52	0.53	0.16	0.18	0.51
22	0.46	0.43	0.38	0.29	0.51	0.52	0.16	0.19	0.50
23	0.46	0.43	0.38	0.28	0.51	0.52	0.16	0.18	0.50
24	0.45	0.42	0.37	0.28	0.50	0.51	0.16	0.19	0.49

6.4.2.2 Analysis of the Results

As shown in the plot diagram of Figure 6.25, there seems to exist certain correlation between *RSR* Metric and relative DOS attackability. It is worthy of digging up further by carrying out relevant statistical analyses to test the existence of the relationship.

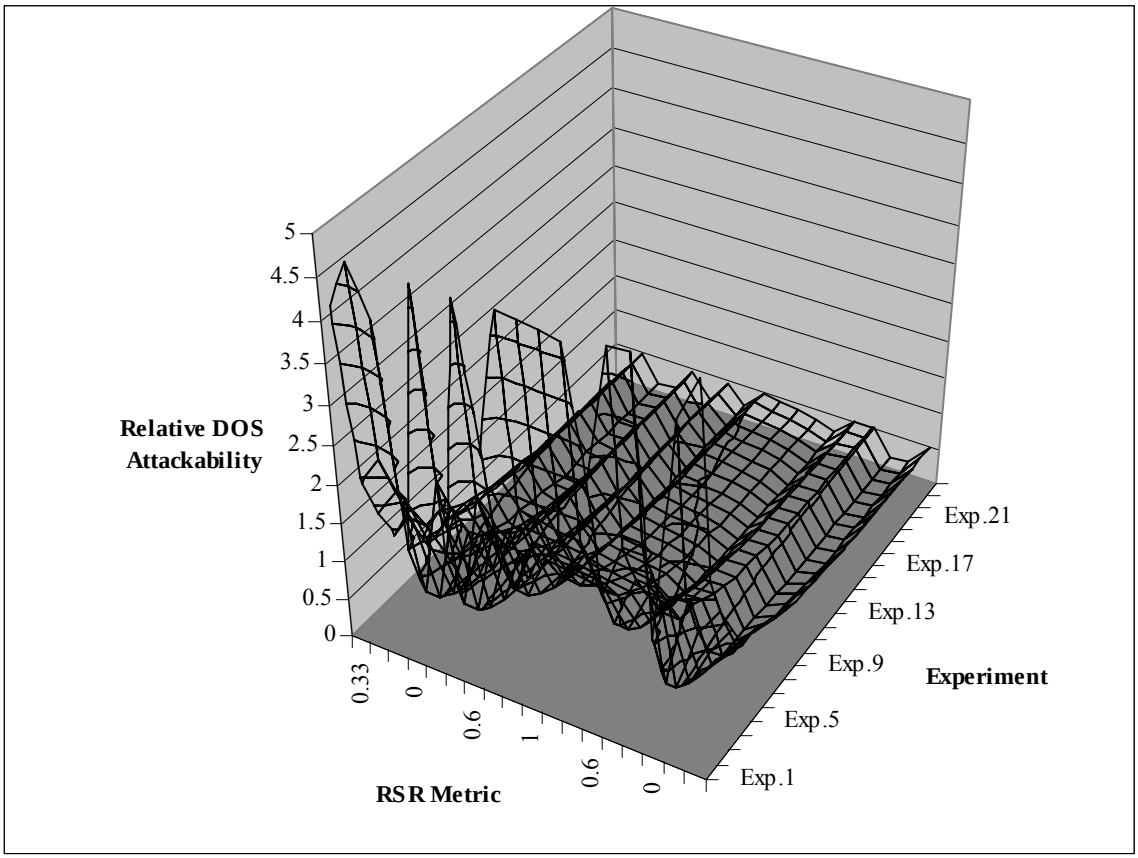

Figure 6.25. Plot Diagram for DOS

At this stage, we only consider the measurement defined in this work to be ordinal scales, accordingly, we choose correlation analysis as the major analysis approach for the data collected in our experiments.

Table 6.9 presents the correlation coefficients between *RSR* metric (provided in Table 6.5) and the relative DOS attackability metric (presented in Table 6.8 (a) and (b)).

Table 6.9 Correlation Coefficients between RSR and Relative DOS Attackability

Experiment No.	***Correlation Coefficients***
1	0.738197
2	0.768678
3	0.762415
4	0.783529
5	0.770721
6	0.787637
7	0.777249
8	0.780151
9	0.782138
10	0.776669
11	0.774821
12	0.769298
13	0.763111
14	0.754333
15	0.749171
16	0.744527
17	0.744199
18	0.741961
19	0.735123
20	0.735123
21	0.736197
22	0.739343
23	0.732107
24	0.735107

Figure 6.26 presents the analysis results of the correlation coefficients based on data provided in Table 6.9. According to Figure 6.26, the mean of the correlation coefficients would be a estimator for the relationship between *RSR* metric and relative DOS attackability. The estimate is around 0.76, which indicates a positive correlation between *RSR* metric and relative DOS attackability. In other words, as the value of service *RSR* metric increases the value of the service relative DOS attackability will increase as well. The median of the correlation coefficients could also be a good indicator. The estimated value is around 0.76, which also signals a strong correlation between the *RSR* metric and relative DOS attackability. The histogram (see Figure 6.26) shows the relative frequencies of all the correlation coefficients from our sample. We observe that most of the correlation coefficients occur in the range 70-80% for this sample.

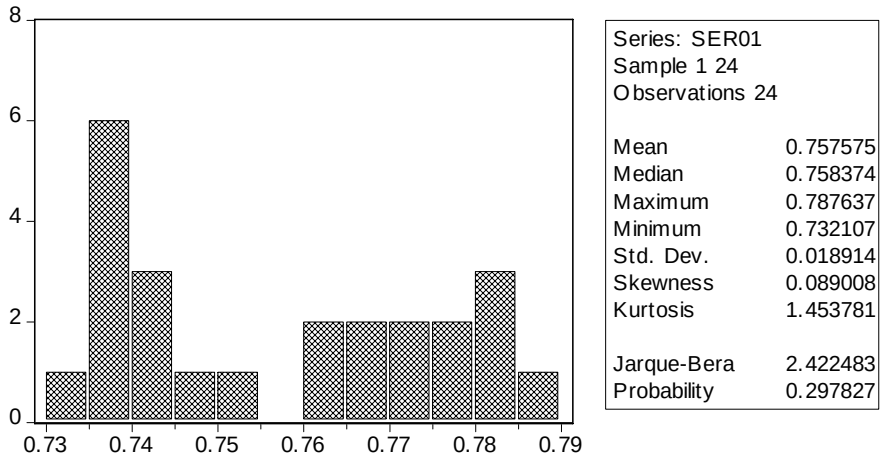

Figure 6.26. Correlation Analysis Results for DOS

In order to further test the strength of the correlation between *RSR* Metric and relative DOS attackability, we apply hypothesis testing and confidence interval analysis.

We start by testing that the hypothesis of the average correlation between the *RSR* Metric and relative DOS attackability is always greater or equal to 0.60, which is considered to be strong correlation. We establish that the null hypothesis is a composite hypothesis and that the alternative hypothesis is one-sided:

$$H_0 : \textit{Average Correlation} \geq 0.60 \qquad \textit{versus} \qquad H_a : \textit{Average Correlation} < 0.60$$

We choose the confidence level to be 0.95, and assume the correlation follows a normal distribution. The appropriate test statistic has a t-distribution. With a sample size of 24, the critical value is -1.71.

Analysis of the collected correlations between *RSR* metric and relative DOS attackability (see Figure 6.26) results in the mean to be 0.76, and the standard deviation to be 0.02, which yields the computed test value to be 39.19. Since the calculated value lies in the acceptance region (39.19>-1.71), we conclude that the correlation between *RSR* metric and relative DOS attackability is strong.

Further study of the strength of the correlation between *RSR* Metric and relative DOS attackability is confidence interval analysis. At the confidence level of 95%, the result of the analysis shows that 95% of time, the true value of the average correlation

between the two variables will be covered by the confidence interval between 0.75 and 0.77 (0.75<correlation<0.77) based on the sample data, and therefore we conclude also that the correlation is strong.

In conclusion, the DOS attackability of a software service and the service's *RSR* metric vary in the same directions: as the *RSR* metric increases, the relative DOS attackability increases. Therefore, we can conclude that at least under the current experimental conditions our hypothesis is supported.

6.5 Empirical Study Using MVN Forum Application

In this section, we repeat the above empirical study using another service-oriented application, namely the MVN Forum application. The motivation to conduct this study is to collect in different application context more empirical evidence for the hypotheses defined in our framework. The study objectives, environment, parameters and procedures remain exactly the same as in the previous study based on the FS application. In section 6.5.1 we describe the MVN Forum application and present its service composition. In Sections 6.5.2 and 6.5.3, we present and analyze the measurement results obtained for URL Jumping attack and Denial of Service attack, respectively.

6.5.1 MVN Forum Overview

MVN forum is an open source bulletin board forum application built on the Java J2EE technology. It is designed and implemented for easy setup and usage. MVN forum like most Internet forum applications provides a platform to allow people to post messages and reply to other people's messages. In addition, the MVN forum provides flexible management services for forum administrators and some privileged users such as individual forum hosts.

The design of the MVN forum application is based on the Model Controller View (MVC) architecture which by default supports a user module and an administrator module. The user module involves typically a controller component which calls a processor component to handle the business logic. More specifically, the processor component

usually maps the user request with specific system operations through a map handler component and subsequently invokes corresponding operations through a web handler component. The web handler component can communicate with the backend database through a data helper component which acts as the interface between web handler component and database. Similarly, the administrator module contains a controller component that forwards administrator's request to an admin processor component. The admin processor component then maps the admin request to admin operations through a map handler component and subsequently calls a web handler component to execute corresponding admin request. Overall, the MVN forum implementation consists of 125 Java classes and 38 JSP pages in addition to the third-party libraries used.

Figure 6.27 depicts the service hierarchy of the MVN forum application where dual circles represent higher level services (referred to as composite services) and single circles represent lowest level services (referred to as atomic services). The services are named after their functionality. The MVN forum application basically provides services for two kinds of users, namely administrators and members. To maintain the forum, the MVN forum administrator as a "super user" needs to execute some administrative services including handling member registration process, managing user groups, sending public emails for maintenance *etc*. Depending on the permissions configured by the forum administrator, a MVN forum member can be granted different level of services including viewing messages, creating new threads and posting replies to existing threads etc. Unlike some other Internet forums where some operational services can be accessed anonymously, the MVN forum application requires authentication and authorization for all of its forum operations.

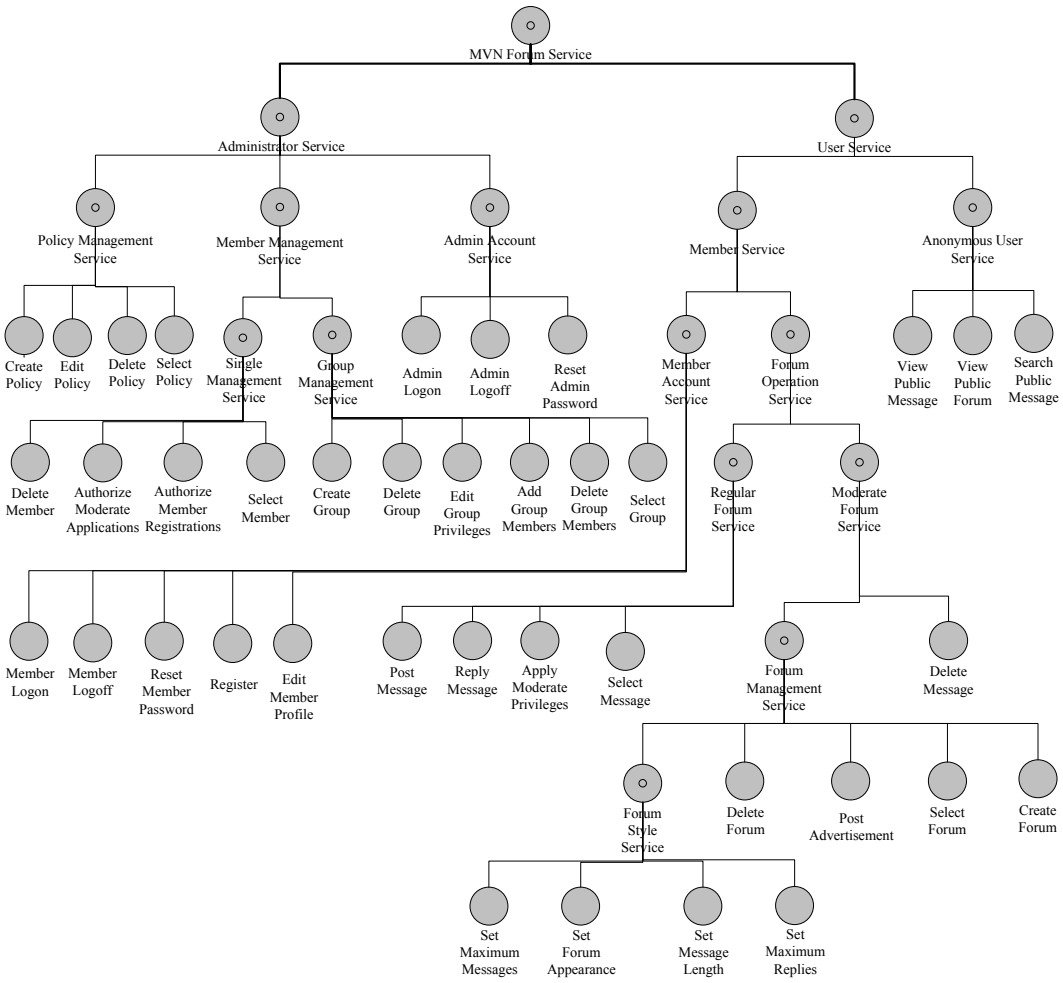

Figure 6.27. Service Hierarchy of MVN Forum Application

6.5.2 Study based on URL Jumping Attack

Like in the previous study, we analyze the service architecture of the MVN forum application to understand which design-level security metrics, in the application context, are likely to bear on its ability to resist to URL Jumping attack. The hypothesis in this study is that *service URL Jumping attackability increases as service complexity increases*. As the metric of service complexity, we use the *ASD* metric defined in chapter 5.

6.5.2.1 Measurement Results

As shown in Figure 6.27, the MVN forum application consists of 16 composite services. In this study, we selected use all these 16 composite services as the target services to validate our hypothesis empirically. We present in the following the attackability measurements obtained, and discuss the corresponding results.

USIE based complexity analysis starts by deriving the USIE configuration graphs of the composite services. Figures 6.28 ~ 6.43 show respectively the USIE configuration graphs constructed for the MVN forum composite services.

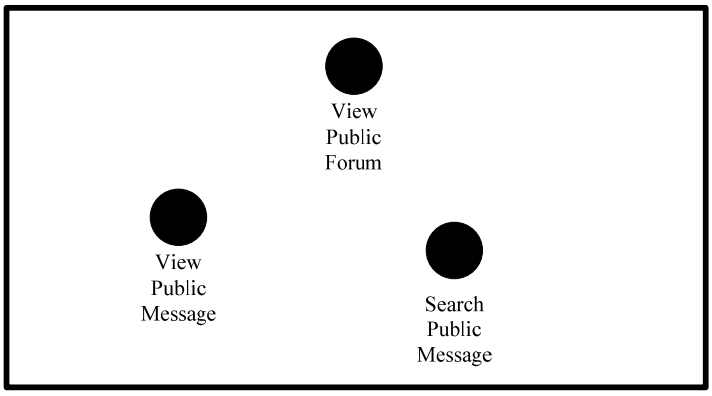

Figure 6.28. The USIE model of Anonymous User Service

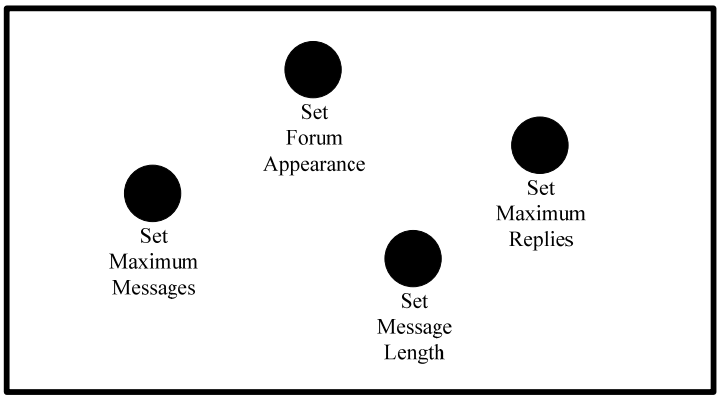

Figure 6.29. The USIE model of Forum Style Service

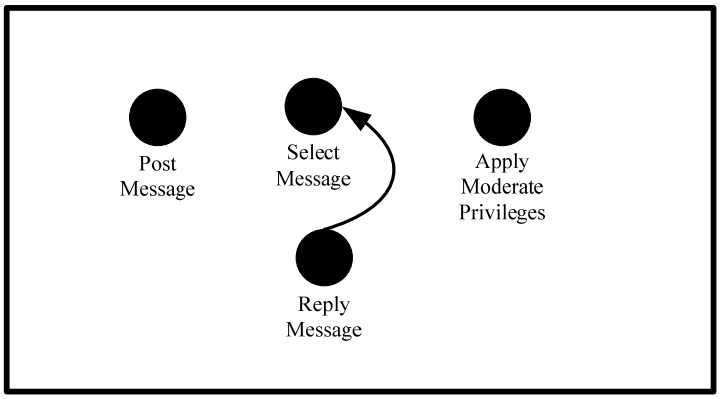

Figure 6.30. The USIE Model of Forum Regular Service

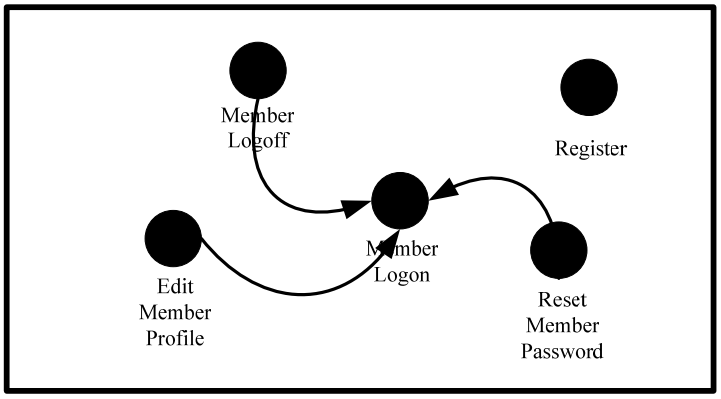

Figure 6.31. The USIE Model of Member Account Service

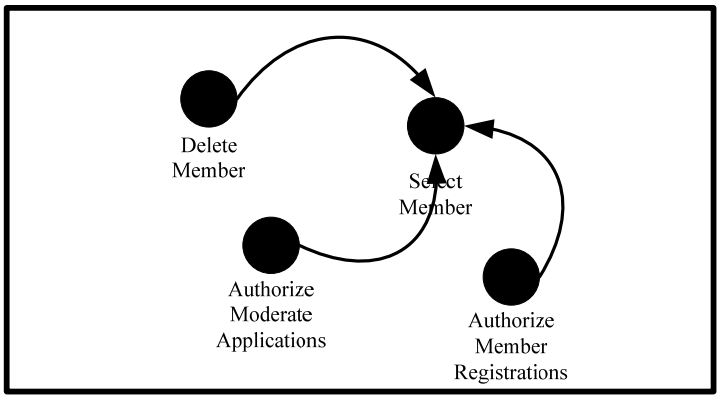

Figure 6.32. The USIE Model of Single Management Service

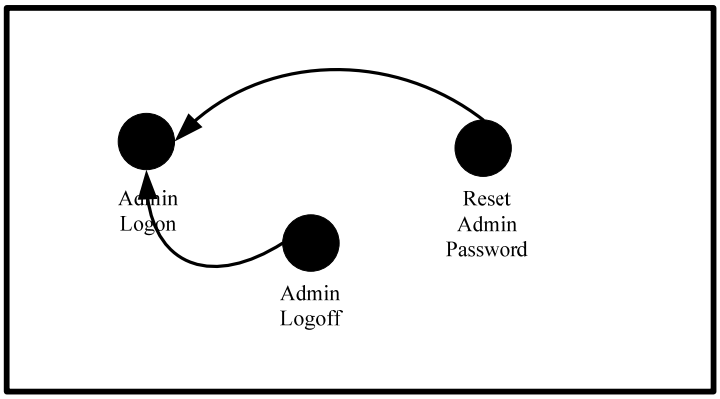

Figure 6.33. The USIE Model of Admin Account Service

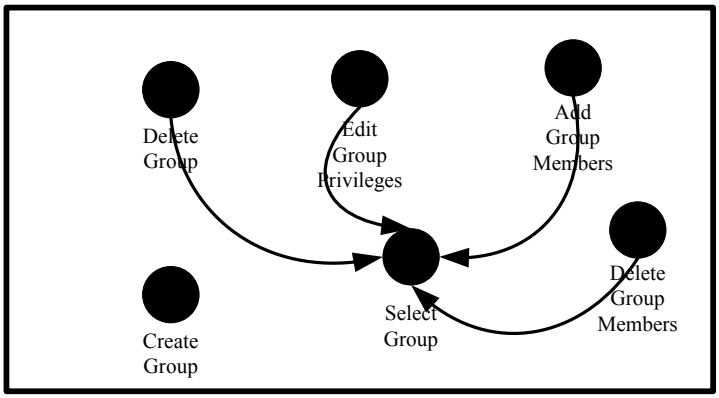

Figure 6.34. The USIE Group Management Service

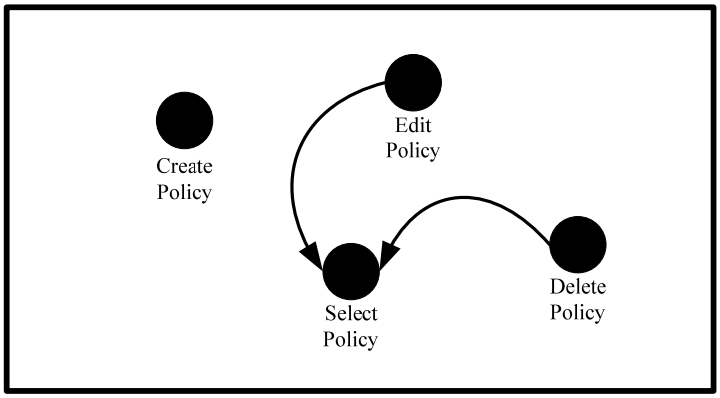

Figure 6.35. The USIE Model of Policy Management Service

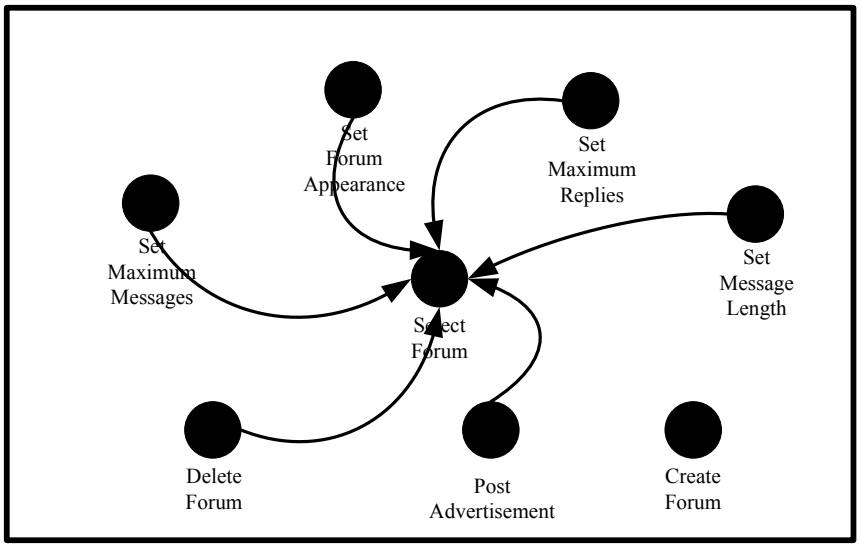

Figure 6.36. The USIE Model of Forum Management Service

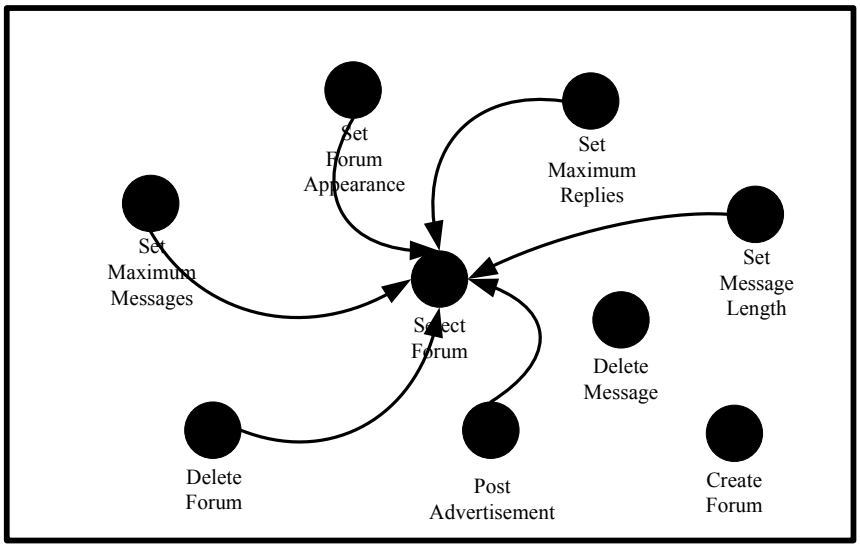

Figure 6.37. The USIE Model of Moderate Forum Service

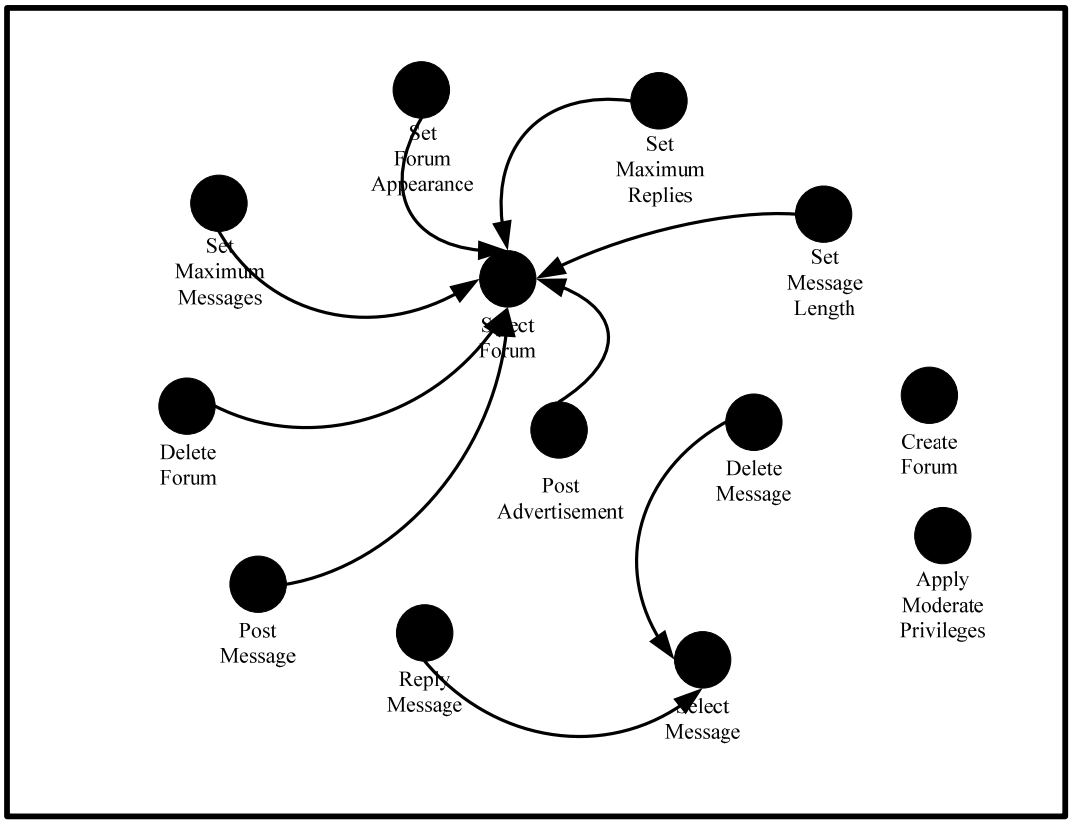

Figure 6.38. The USIE Model of Forum Operation Service

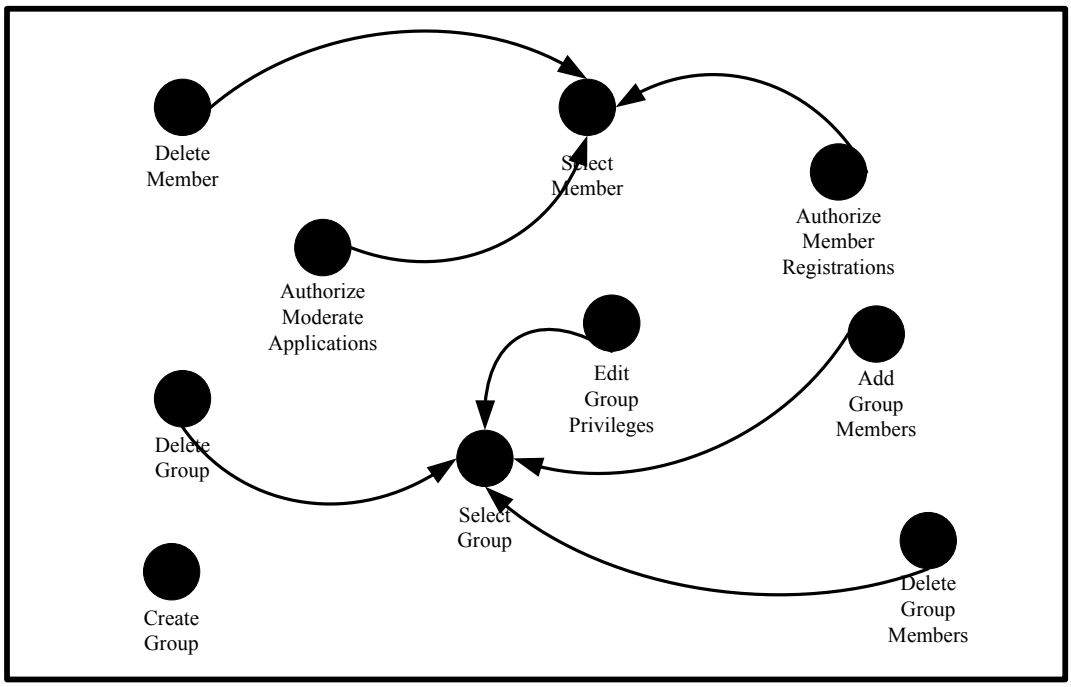

Figure 6.39. The USIE Model of Member Management Service

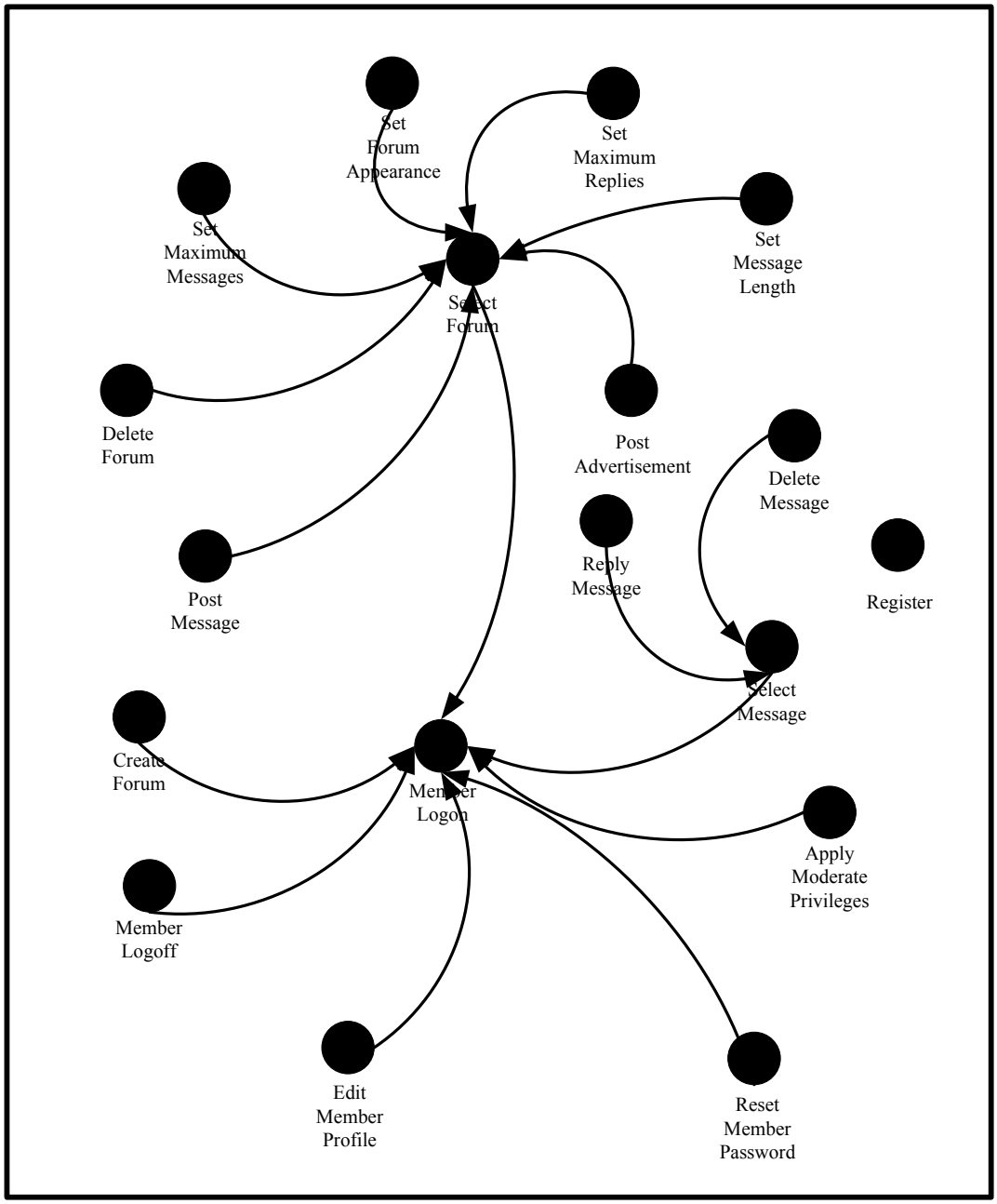

Figure 6.40. The USIE Model of Member Service

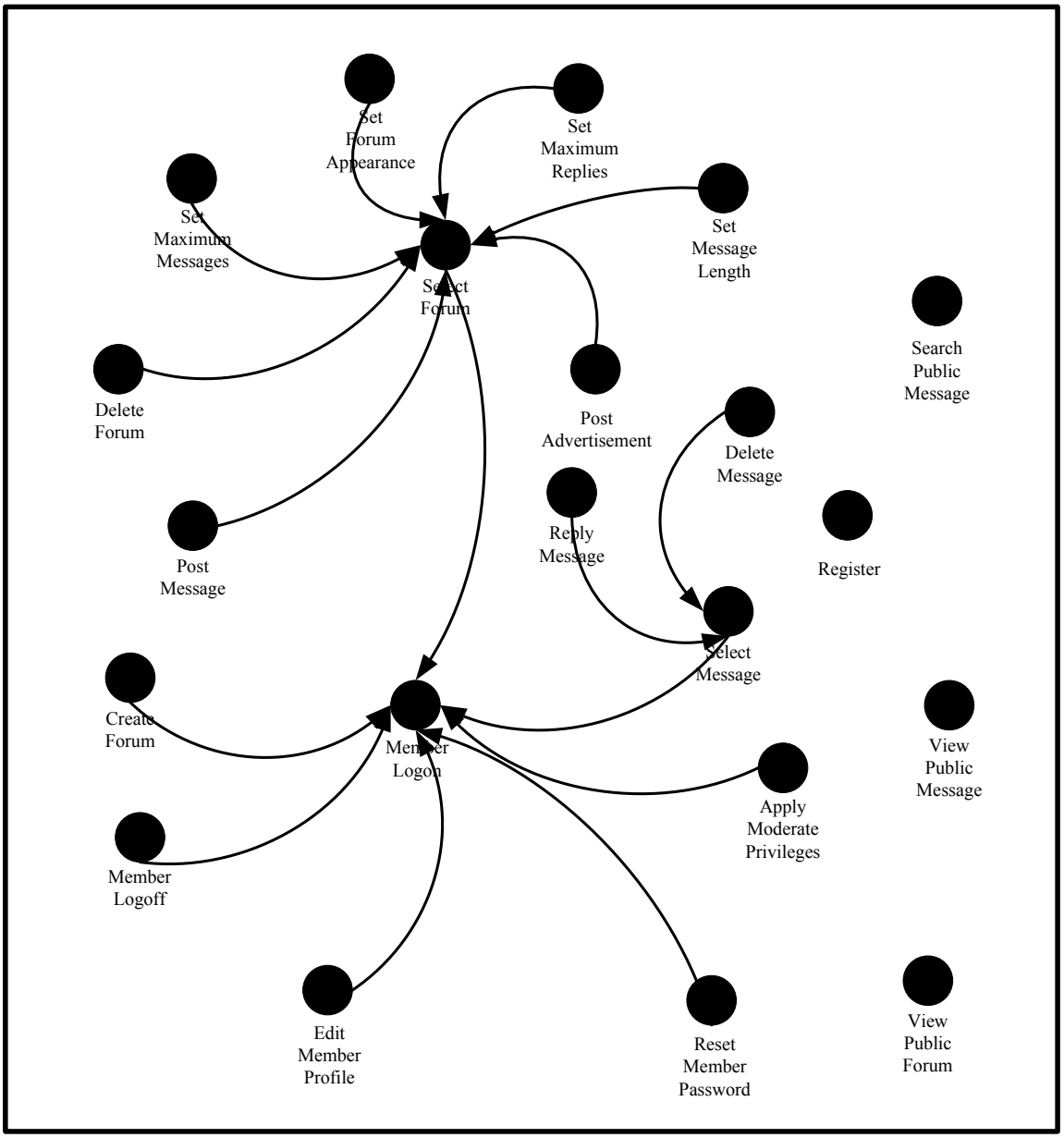

Figure 6.41. The USIE Model of User Service

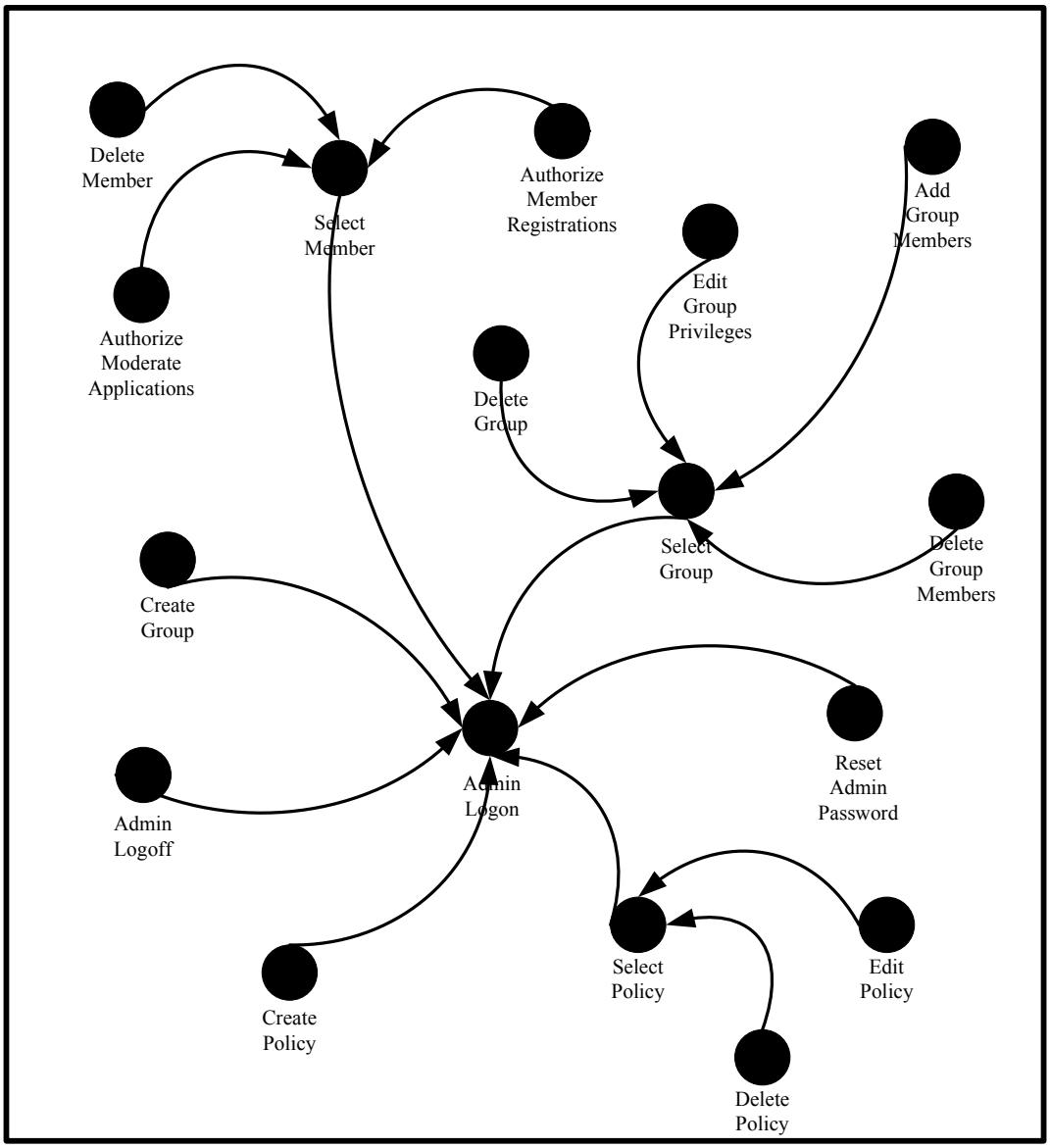

Figure 6.42. The USIE Model of Administrator Service

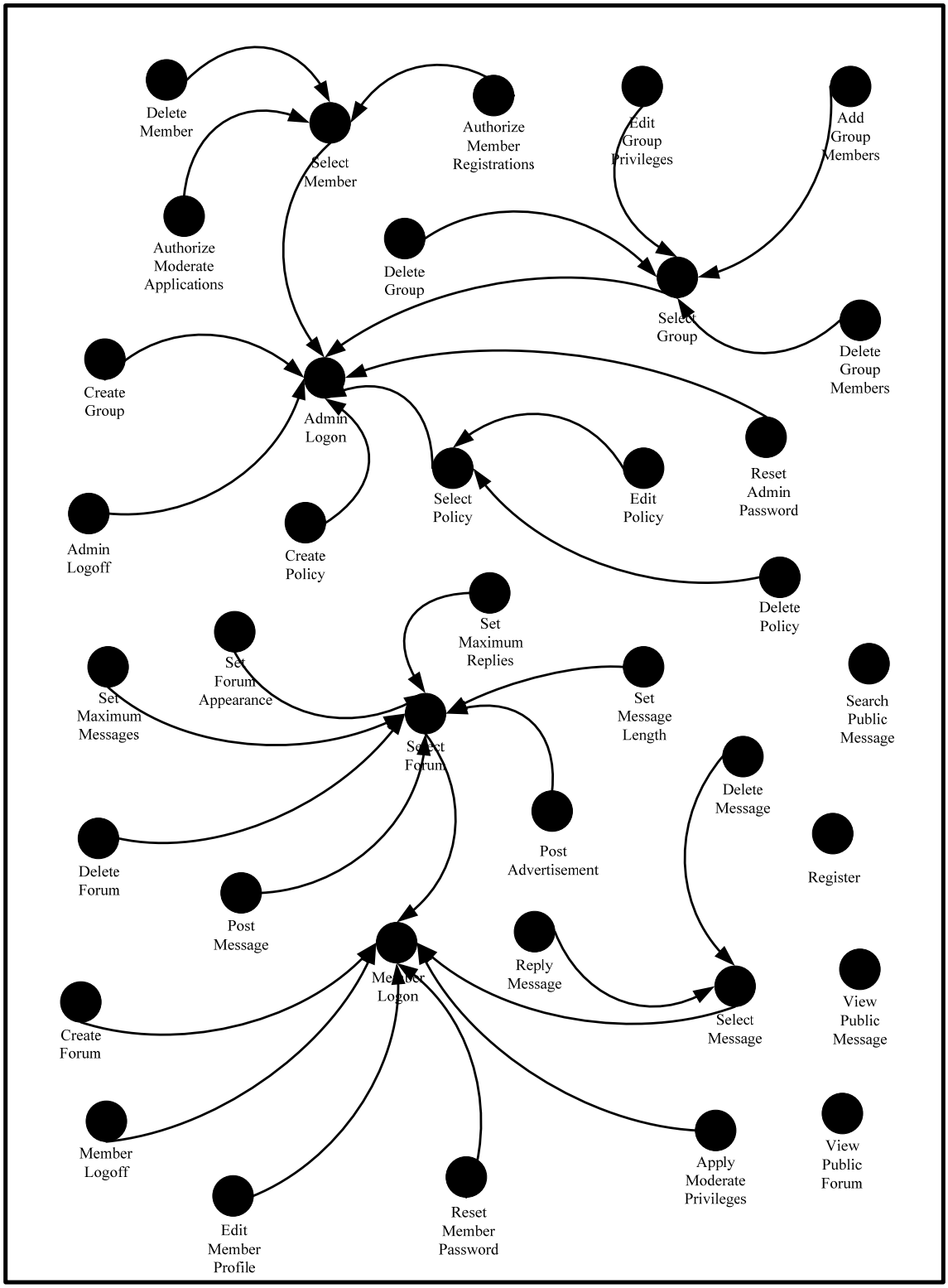

Figure 6.43. The USIE Model of MVN Forum Service

Based on the configuration graphs of MVN Forum composite services, we can compute complexity metrics based on the metric definition introduced in section 5.1.3. Table 6.10 shows the corresponding *ASD* values for each of the composite services involved in the forum application.

Table 6.10 Service ASD Metric Values

No.	***Composite Service*** $\bar{s}$	***ASD(*** $\bar{s}$ ***)***
1	Anonymous User Service	0
2	Forum Style Service	0
3	Forum Regular Service	0.25
4	Member Account Service	0.6
5	Single Management Service	0.75
6	Admin Account Service	0.67
7	Group Management Service	0.67
8	Policy Management Service	0.5
9	Forum Management Service	0.75
10	Moderate Forum Service	0.67
11	Forum Operation Service	0.71
12	Member Management Service	0.70
13	Member Service	1.39
14	User Service	1.19
15	Administrator Service	1.47
16	MVN Forum Service	1.32

Like in the previous study with FS, we conduct the URL Jumping experiments over 20 different runs. In each independent run of the experiments, we randomly choose URLs from the identified URL sets of each target services to look for URL Jumping vulnerability. If no URL Jumping vulnerability is found for a target composite service after attacking all of its atomic services, we consider the corresponding attack effort to be infinite.

We compare relative service attackability under the same attack reward by posing *AttackReward = 1*. Therefore, in each run of the experiments, the attacks on the corresponding composite services will stop once we identify successfully URL Jumping vulnerability for the corresponding composite services. Table 6.11 (a) and (b) present the results of the URL Jumping attacks over the 20 independent runs. Table 6.12 (a) and (b), which are derived from Table 6.11 (a) and (b), present the relative URL Jumping attackability for the composite services in each of the corresponding runs.

Table 6.11 (a) URL Jumping Attack Effort under *AttackReward = 1*

	Experiment No.									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Anonymous User Service	1	2	2	2	1	2	2	1	2	2
Forum Style Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Forum Regular Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Member Account Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Single Management Service	5	4	3	5	5	5	4	4	3	5
Admin Account Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Group Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Policy Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Forum Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Moderate Forum Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Forum Operation Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Member Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Member Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
User Service	3	7	6	8	9	7	2	2	5	4
Administrator Service	5	15	10	10	11	8	5	4	7	6
MVN Forum Service	3	6	6	7	10	6	2	2	5	4

Table 6.11 (b) URL Jumping Attack Effort under *AttackReward = 1* (continued)

	Experiment No.									
	11	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
Anonymous User Service	1	2	2	1	2	1	2	2	2	1
Forum Style Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Forum Regular Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Member Account Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Single Management Service	4	3	4	4	4	5	5	5	5	5
Admin Account Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Group Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Policy Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Forum Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Moderate Forum Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Forum Operation Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Member Management Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Member Service	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
User Service	8	4	13	5	4	6	7	8	7	13
Administrator Service	10	6	14	9	6	8	8	11	12	15
MVN Forum Service	8	5	12	8	4	7	7	9	8	12

Table 6.12 (a) Relative URL Jumping Attackability

	Experiment No.									
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Anonymous User Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Forum Style Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Forum Regular Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Member Account Service	0.20	0.25	0.33	0.20	0.20	0.20	0.25	0.25	0.33	0.20
Single Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Admin Account Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Group Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Policy Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Forum Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Moderate Forum Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Forum Operation Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Member Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Member Service	0.33	0.14	0.17	0.13	0.11	0.14	0.50	0.50	0.20	0.25
User Service	0.20	0.07	0.10	0.10	0.09	0.13	0.20	0.25	0.14	0.17
Administrator Service	0.33	0.17	0.17	0.14	0.10	0.17	0.50	0.50	0.20	0.25
MVN Forum Service	0.25	0.13	0.13	0.25	0.13	0.25	0.25	0.33	0.17	0.20

Table 6.12 (b) Relative URL Jumping Attackability (continued)

	Experiment No.									
	11	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
Anonymous User Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Forum Style Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Forum Regular Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Member Account Service	0.25	0.33	0.25	0.25	0.25	0.20	0.20	0.20	0.20	0.20
Single Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Admin Account Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Group Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Policy Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Forum Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Moderate Forum Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Forum Operation Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Member Management Service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Member Service	0.13	0.25	0.08	0.20	0.25	0.17	0.14	0.13	0.14	0.08
User Service	0.10	0.17	0.07	0.11	0.17	0.13	0.13	0.09	0.08	0.07
Administrator Service	0.13	0.20	0.08	0.13	0.25	0.14	0.14	0.11	0.13	0.08
MVN Forum Service	0.20	0.33	0.08	0.14	0.25	0.11	0.25	0.11	0.13	0.07

6.5.2.2 Analysis of the Results

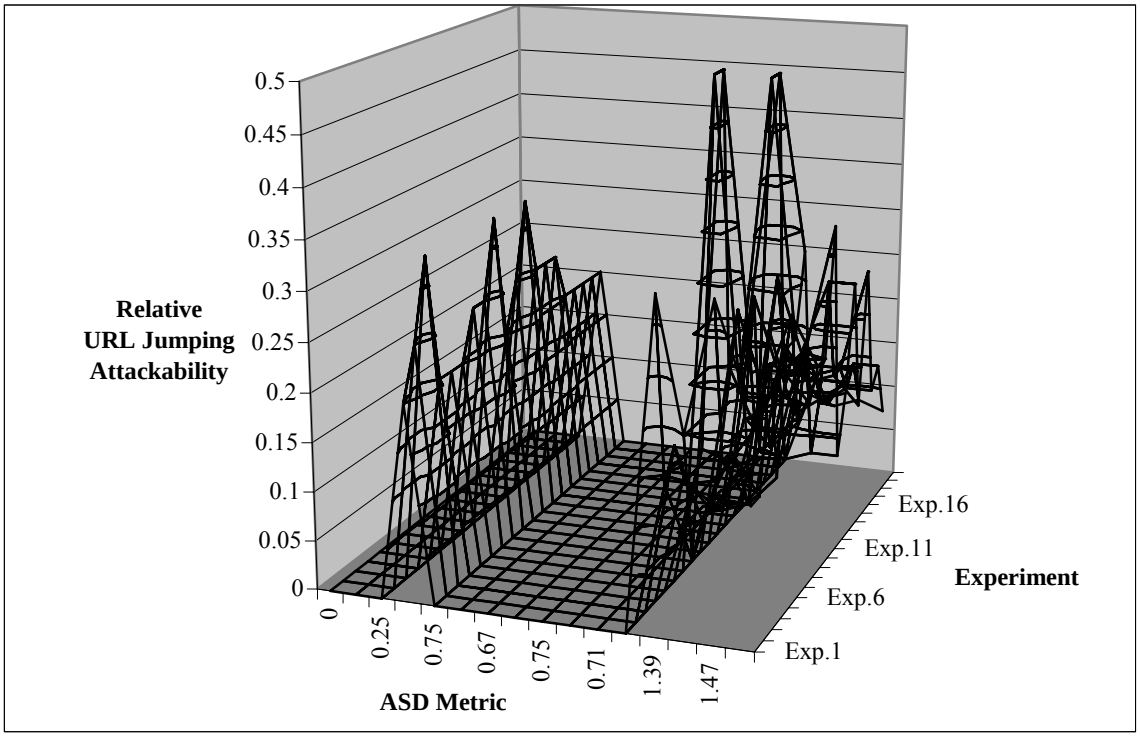

Figure 6.44. Plot Diagram for URL Jumping

Table 6.13 Correlation Coefficients between ASD Metric and Relative URL Jumping Attackability

Experiment No.	***Correlation Coefficients***
1	0.789695
2	0.551720
3	0.483335
4	0.640670
5	0.561826
6	0.680948
7	0.766000
8	0.793518
9	0.570063
10	0.758556
11	0.560329
12	0.639658
13	0.360996
14	0.586715
15	0.725765
16	0.645139
17	0.664408
18	0.576733
19	0.607003
20	0.433623

As shown in the plot diagram of Figure 6.44, there seems to exist certain correlation between the *ASD* Metric and relative URL Jumping Attackability. It is worthy of digging up further by carrying out relevant statistical analyses to test the existence of the relationship.

As mentioned above, we only consider the measurement defined in this work to be at the ordinal scales level, accordingly, we choose correlation analysis as the major analysis approach for the data collected in our experiments.

Table 6.13 presents the correlation coefficients between service *ASD* metrics (presented in Table 6.10) and the service relative URL Jumping attackability metrics (presented in Table 6.12)

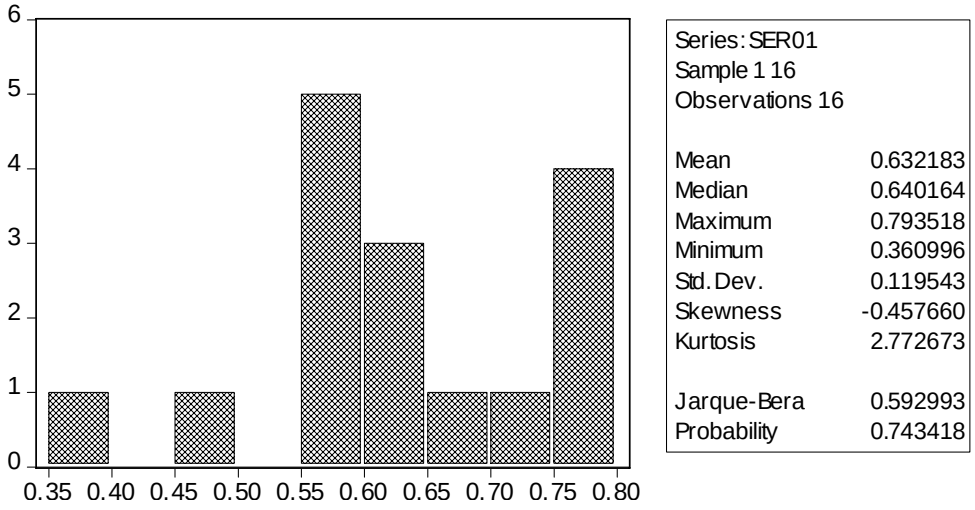

Figure 6.45. Analysis of Correlation Coefficients for URL Jumping

Figure 6.45 presents the results of the analysis of the correlation coefficients. The estimate of the mean of the correlation coefficients is around 0.63, which indicates a positive correlation between *ASD* metric and relativeURL Jumping attackability. The estimated value for the median of the correlation coefficients is around 0.64, which also indicates stronger correlation between the *ASD* metric and relative URL Jumping attackability. Furthermore from the histogram (see Figure 6.45) we observe that most of the correlation coefficients fall in the range 55-80% for this sample.

Like previously, we conduct further test of the strength of the correlation between *ASD* Metric and relative URL Jumping attackability through hypothesis testing and confidence interval analysis.

We begin with testing that the hypothesis of the correlation between the *ASD* Metric and relative URL Jumping attackability is always greater or equal to 0.60 on average, which is considered to be strong correlation. We establish that the null hypothesis is a composite hypothesis and that the alternative hypothesis is one-sided:

$$H_0 : \textit{Average Correlation} \geq 0.60 \quad \textit{versus} \quad H_a : \textit{Average Correlation} < 0.60$$

We choose the confidence level to be 0.95, and assume the correlation follows a normal distribution. The appropriate test statistic has a t-distribution. With a sample size of 20, the critical value is -1.73.

Analysis of the collected correlations between *ASD* Metric and relative URL Jumping attackability (in Figure 6.45) results in the mean to be 0.63, and the standard deviation to be 0.12, which yields the computed test value to be 1.12. Since the calculated value lies in the acceptance region (1.12>-1.73), we conclude that the average correlation between *ASD* Metric and relative URL Jumping attackability is strong.

The result of the confidence interval analysis shows that 95% of time, the true value of the average correlation between the two variables will be covered by the confidence interval between 0.57 and 0.69 (0.57<correlation<0.69), and therefore we conclude that the correlation is strong.

In conclusion, like in the previous study based on FS application, the URL Jumping attackability value increases as the *ASD* metric increases, which suggests that, at least under the current experimental conditions our hypothesis is supported.

6.5.3 Study based on Application DOS Attack

Like in the study based on FS application, our objective here is to analyze the service architecture of the MVN Forum application to understand which design-level security

metrics, in the application context, are likely to bear on its ability to resist to application DOS attack. Our study hypothesis is that *application DOS attackability increases as the service coupling between target services and victim services increases*. We use the *RSR* metric defined in Chapter 5 as the service coupling metric in this study.

6.5.3.1 Measurement Results

In this study, we selected 7 atomic services from the MVN Forum application as the context to evaluate empirically our hypothesis. The selected services include *Member Login, Register, Post Message, Reply Message, Search Public Message, Create Forum*, and *Edit Member Profile*. Figure 6.46 ~ 6.52 illustrate the USIE models for these atomic services. Each of the atomic service involves multiple operations as shown in the Figures.

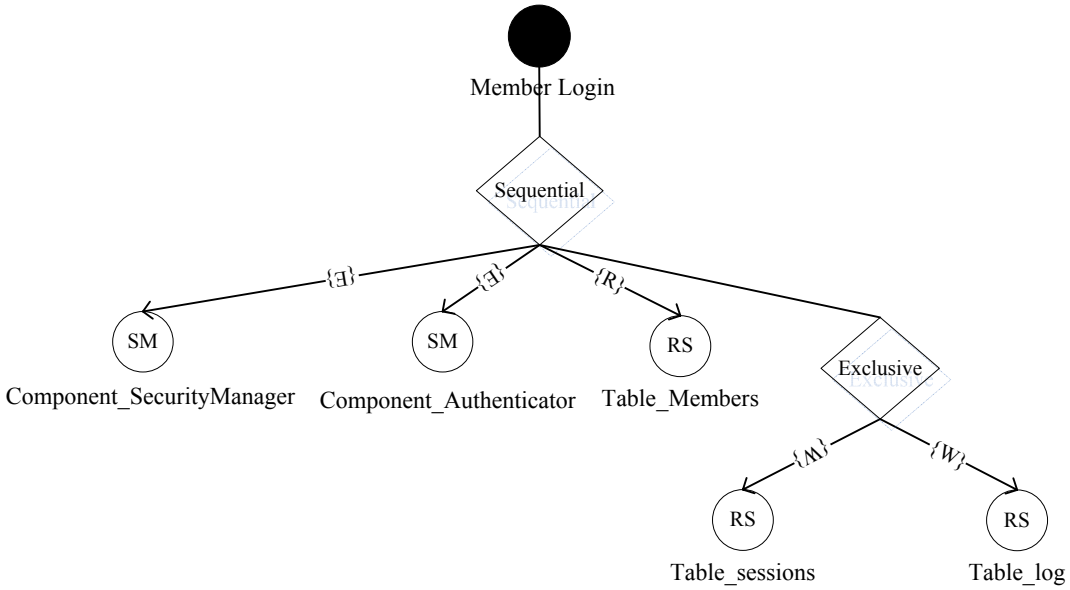

Figure 6.46. USIE Graph for the Atomic Service *Member Login*

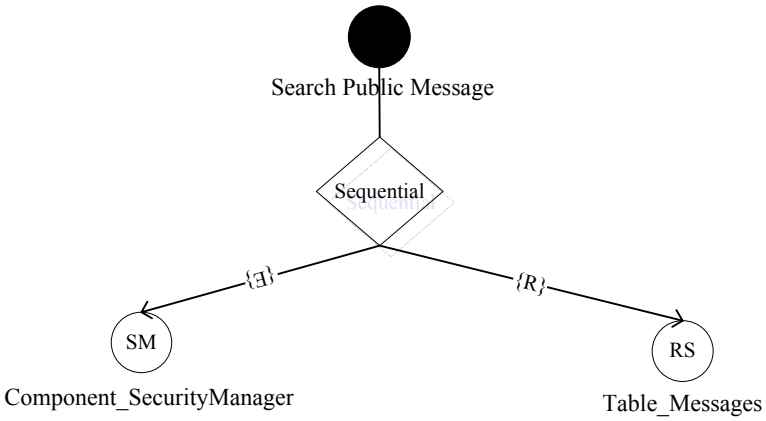

Figure 6.47. USIE Graph for the Atomic Service *Search Public Message*

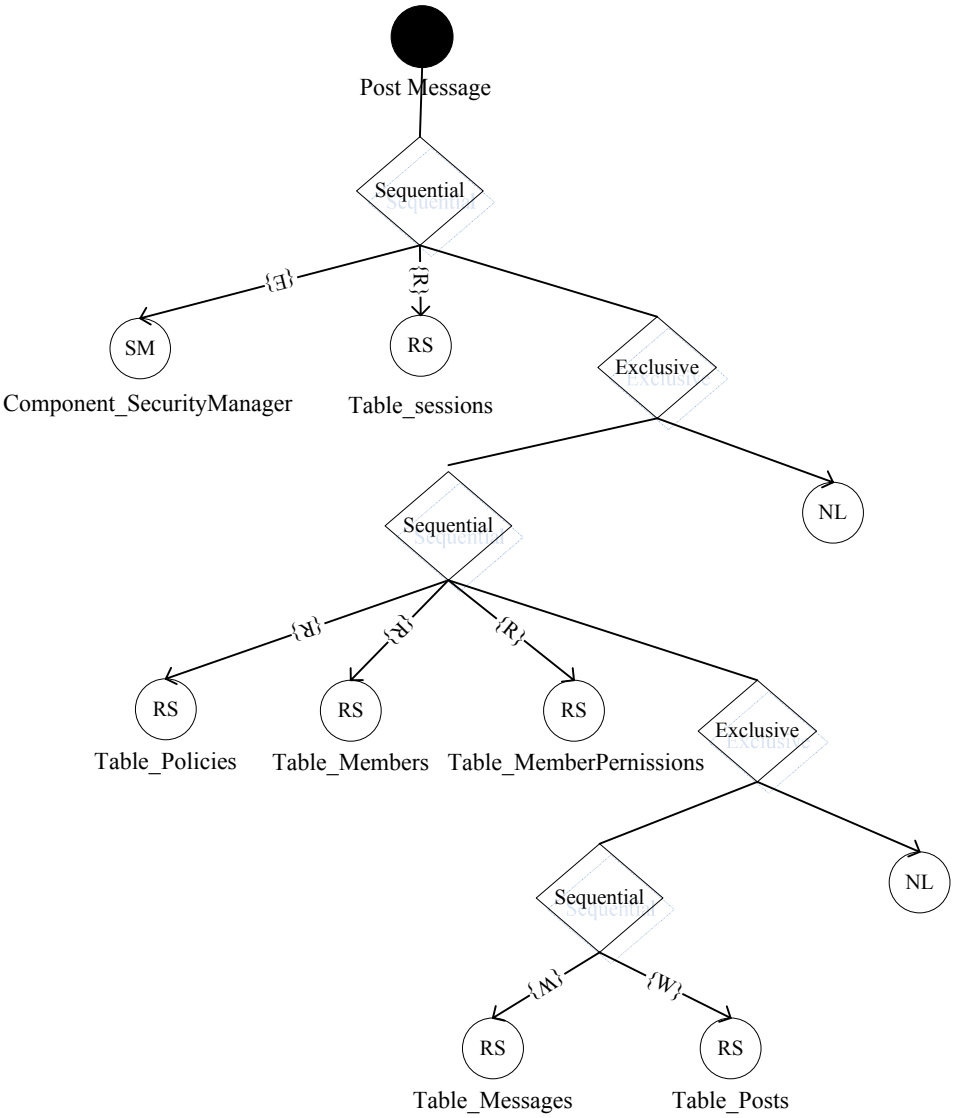

Figure 6.48. USIE Graph for the Atomic Service *Post Message*

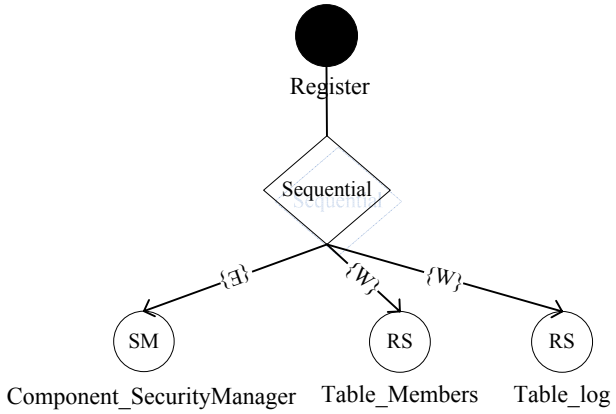

Figure 6.49. USIE Graph for the Atomic Service *Register*

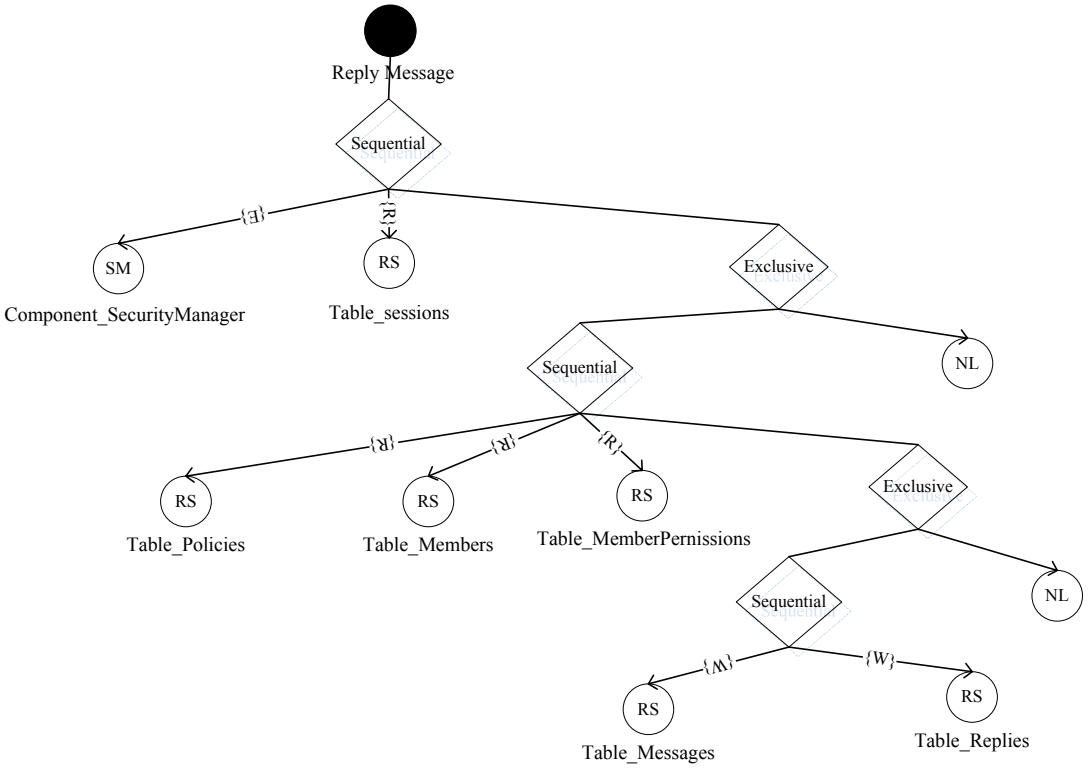

Figure 6.50. USIE Graph for the Atomic Service *Reply Message*

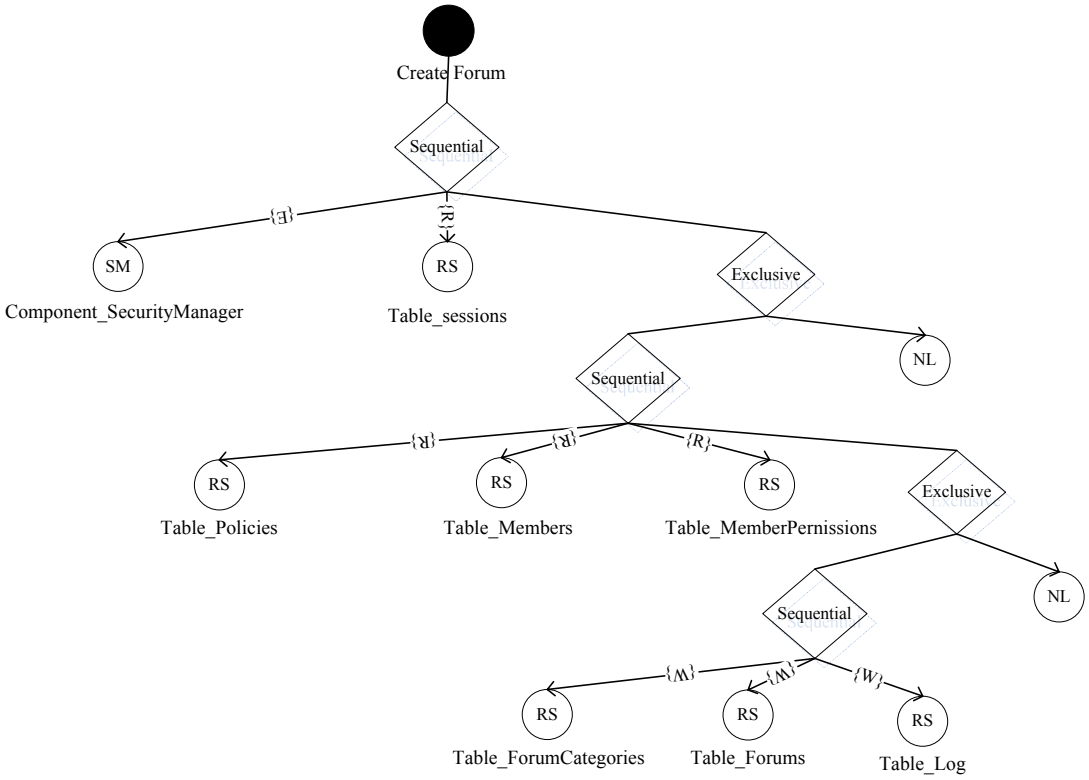

Figure 6.51. USIE Graph for the Atomic Service *Create Forum*

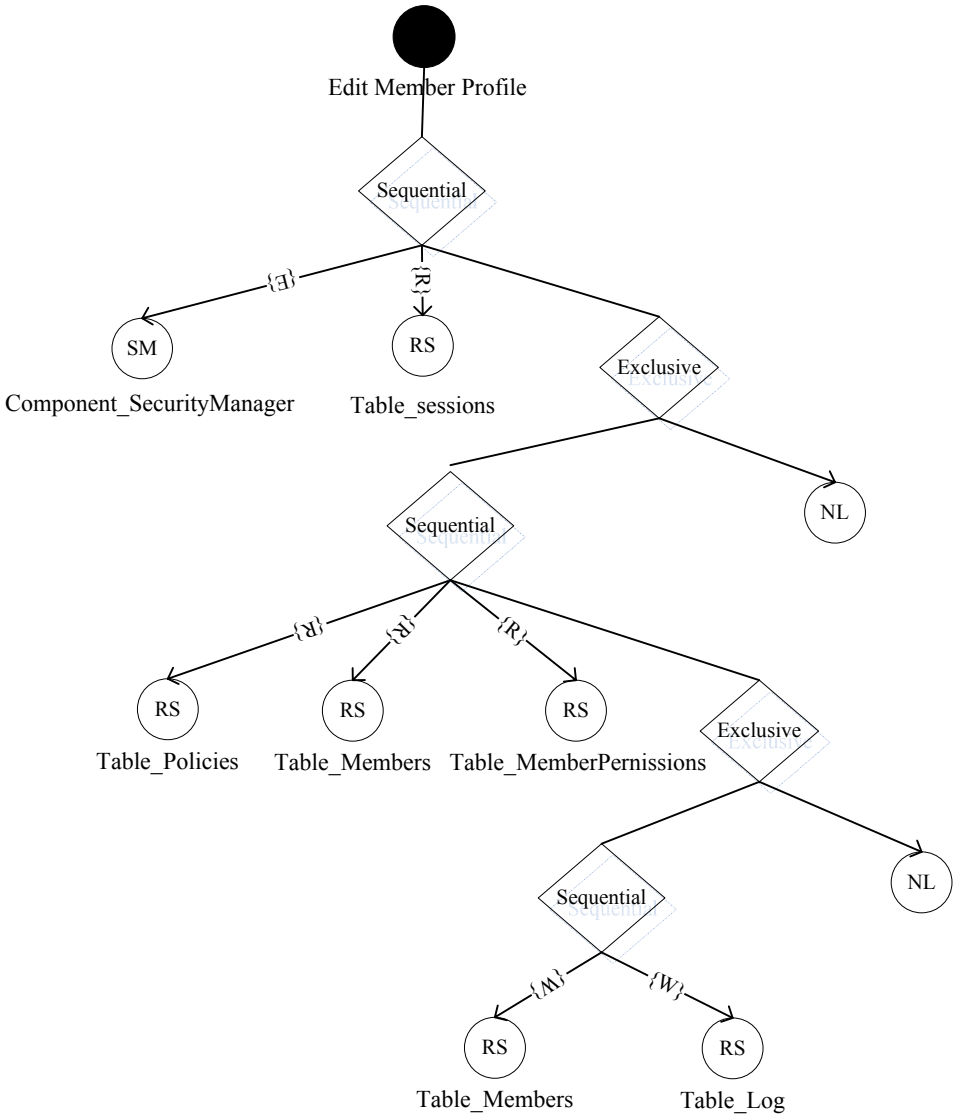

Figure 6.52. USIE Graph for the Atomic Service *Edit Member Profile*

Like in FS application study, we computed the *RSR* metric on pairs of atomic services involved in this study. We selected randomly 18 pairs of atomic services from the atomic services involved in this study, each pair consisting of a target service and a possible victim service. Based on Figures 6.46 ~ 6.52, we computed coupling metrics using the *RSR* metrics for each of the atomic services pairs. Table 6.14 lists the corresponding measurement results.

Table 6.14 RSR Values

No.	*Target Service*	*Victim Service*	*RSR($U_{targetService}$,$U_{victimService}$)*
1	Register	Member Login	0.67
2	Post Message	Reply Message	0.71
3	Post Message	Edit Member Profile	0.5
4	Post Message	Create Forum	0.44
5	Reply Message	Create Forum	0.44
6	Post Message	Member Login	0.29
7	Post Message	Register	0.14
8	Register	Edit Member Profile	0.33
9	Edit Member Profile	Member Login	0.5
10	Edit Member Profile	Reply Message	0.5
11	Post Message	Post Message	1
12	Edit Member Profile	Create Forum	0.63
13	Reply Message	Member Login	0.29
14	Create Forum	Member Login	0.43
15	Create Forum	Register	0.29
16	Register	Reply Message	0.14
17	Post Message	Search Public Message	0.17
18	Reply Message	Search Public Message	0.17

DOS attacks were conducted using the same attack configuration and script as in the case of the FS application.

Table 6.15 Regular Response Times of Atomic Services: *Workload = 1 Request/Second*

No.	***Atomic Service***	***Regular Response Time (second)***
1	Register	0.21
2	Member Login	0.13
3	Post Message	0.25
4	Reply Message	0.22
5	Search Public Message	0.15
6	Edit Member Profile	0.19
7	Create Forum	0.27

We set the regular workload used in our attackability measurements to 1 *request/second*. To capture the regular response time corresponding to the regular workload for each of the atomic services, we simply run the attack scripts under regular workload for each of the atomic services and observe the corresponding response time.

Table 6.15 presents the regular response time observed for the atomic services involved in this study.

We conducted 24 rounds of DOS attacks by increasing the attack workload in each round. In each round, each of the target atomic services selected in our study was submitted to the same attack workload, and then the attack reward was computed for the corresponding victim service. Table 6.16 (a) and (b) summarize the measurement results. Table 6.17 (a) and (b) list the relative DOS attackability computed from the measurements listed in Table 6.16 (a) and (b).

Table 6.16 (a) Measurements for DOS Attack Experiments: The columns numbered from 1 to 9 list the attack rewards corresponding to the atomic service pairs numbered from 1 to 9 in Table 6.14.

Attack Session No.	***Attack Effort***	***Attack Reward***								
		1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
1	*2*	11.54	11.57	10.10	9.24	11.47	3.74	3.78	7.51	10.13
2	*4*	11.18	11.23	9.12	9.48	11.04	3.89	3.95	7.54	9.17
3	*6*	12.67	12.74	9.72	9.54	12.46	3.96	4.05	7.49	9.79
4	*8*	13.15	13.25	8.64	9.60	12.86	4.03	4.16	7.39	8.74
5	*10*	11.76	11.88	8.16	9.60	13.32	4.08	4.24	7.44	11.64
6	*12*	14.40	14.54	7.49	10.08	13.97	4.32	4.51	7.63	7.63
7	*14*	14.95	15.12	9.07	9.24	14.45	4.54	4.25	7.90	6.72
8	*16*	15.17	15.36	8.45	11.04	14.59	4.80	5.05	8.06	8.64
9	*18*	15.77	15.98	7.99	11.34	15.12	4.97	5.25	8.42	8.21
10	*20*	16.08	16.32	7.20	11.40	15.36	5.04	5.35	8.40	7.44
11	*22*	16.90	17.16	7.92	11.88	16.10	5.28	5.62	8.71	8.18
12	*24*	17.57	17.86	7.20	12.24	16.70	5.47	5.85	8.93	7.49
13	*26*	18.41	18.72	6.86	12.48	17.47	5.62	6.02	9.05	7.18
14	*28*	18.82	19.15	6.05	12.60	17.81	5.71	6.15	9.07	6.38
15	*30*	19.80	20.16	5.40	12.60	18.72	5.76	6.23	9.36	5.76
16	*32*	20.35	20.74	6.14	13.44	19.20	6.14	6.64	9.22	6.53
17	*34*	22.03	22.44	6.53	14.28	20.81	6.53	7.06	9.79	6.94
18	*36*	22.90	23.33	6.48	15.12	21.60	6.91	7.47	10.37	6.91
29	*38*	23.71	24.17	6.38	15.96	22.34	7.30	7.89	10.49	6.84
20	*40*	24.96	25.44	6.72	16.80	23.52	7.68	8.30	11.04	7.20
21	*42*	25.70	26.21	6.05	16.38	24.19	7.56	8.22	11.59	6.55
22	*44*	26.40	26.93	6.86	17.16	24.82	7.92	8.61	12.14	7.39
23	*46*	27.60	28.15	6.62	17.94	25.94	8.28	9.00	12.14	7.18
24	*50*	29.40	30.00	7.80	19.50	27.60	9.00	9.78	13.20	8.40

Table 6.16 (b) Measurements for DOS Attack Experiments (continued): The columns numbered from 10 to 18 list the attack rewards corresponding to the atomic service pairs numbered from 10 to 18 in Table 6.14.

Attack Session No.	***Attack Effort***	***Attack Reward***								
		10	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
1	*2*	11.45	11.59	11.26	7.66	11.38	11.52	3.70	10.25	8.66
2	*4*	10.99	11.28	10.61	7.82	10.85	11.14	3.79	9.41	8.39
3	*6*	12.38	12.82	11.81	7.92	12.17	12.60	3.82	10.15	9.50
4	*8*	12.77	13.34	12.00	7.97	12.48	13.06	3.84	9.22	9.86
5	*10*	11.28	12.00	12.00	8.16	12.60	11.64	3.84	8.88	8.82
6	*12*	13.82	14.69	12.67	8.50	13.39	14.26	4.03	8.35	10.80
7	*14*	14.28	15.29	12.77	8.90	13.61	14.78	3.70	10.08	11.21
8	*16*	14.40	15.55	12.86	9.22	13.82	14.98	4.42	9.60	11.38
9	*18*	14.90	16.20	13.18	9.72	14.26	15.55	4.54	9.29	11.83
10	*20*	15.12	16.56	13.20	9.84	14.40	15.84	4.56	8.64	12.06
11	*22*	15.84	17.42	13.73	10.30	15.05	16.63	4.75	9.50	12.67
12	*24*	16.42	18.14	14.11	10.66	15.55	17.28	4.90	8.93	13.18
13	*26*	17.16	19.03	14.66	10.92	16.22	18.10	4.99	8.74	13.81
14	*28*	17.47	19.49	14.78	11.09	16.46	18.48	5.04	8.06	14.11
15	*30*	18.36	20.52	15.48	11.52	17.28	19.44	5.04	7.56	14.85
16	*32*	18.82	21.12	15.74	11.52	17.66	19.97	5.38	8.45	15.26
17	*34*	20.40	22.85	17.14	12.24	19.18	21.62	5.71	8.98	16.52
18	*36*	21.17	23.76	17.71	12.96	19.87	22.46	6.05	9.07	17.17
29	*38*	21.89	24.62	18.24	13.22	20.52	23.26	6.38	9.12	17.78
20	*40*	23.04	25.92	19.20	13.92	21.60	24.48	6.72	9.60	18.72
21	*42*	23.69	26.71	19.66	14.62	22.18	25.20	6.55	9.07	19.28
22	*44*	24.29	27.46	20.06	15.31	22.70	25.87	6.86	10.03	19.80
23	*46*	25.39	28.70	20.98	15.46	23.74	27.05	7.18	9.94	20.70
24	*50*	27.00	30.60	22.20	16.80	25.20	28.80	7.80	11.40	22.05

Table 6.17 (a) Relative DOS Attackability Values derived from Table 6.16 (a).

Attack Session No.	***Relative DOS Attackability***								
	1	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
1	5.77	5.78	5.05	4.62	5.74	1.87	1.89	3.76	5.06
2	2.80	2.81	2.28	2.37	2.76	0.97	0.99	1.88	2.29
3	2.11	2.12	1.62	1.59	2.08	0.66	0.68	1.25	1.63
4	1.64	1.66	1.08	1.20	1.61	0.50	0.52	0.92	1.09
5	1.18	1.19	0.82	0.96	1.33	0.41	0.42	0.74	1.16
6	1.20	1.21	0.62	0.84	1.16	0.36	0.38	0.64	0.64
7	1.07	1.08	0.65	0.66	1.03	0.32	0.30	0.56	0.48
8	0.95	0.96	0.53	0.69	0.91	0.30	0.32	0.50	0.54
9	0.88	0.89	0.44	0.63	0.84	0.28	0.29	0.47	0.46
10	0.80	0.82	0.36	0.57	0.77	0.25	0.27	0.42	0.37
11	0.77	0.78	0.36	0.54	0.73	0.24	0.26	0.40	0.37
12	0.73	0.74	0.30	0.51	0.70	0.23	0.24	0.37	0.31
13	0.71	0.72	0.26	0.48	0.67	0.22	0.23	0.35	0.28
14	0.67	0.68	0.22	0.45	0.64	0.20	0.22	0.32	0.23
15	0.66	0.67	0.18	0.42	0.62	0.19	0.21	0.31	0.19
16	0.64	0.65	0.19	0.42	0.60	0.19	0.21	0.29	0.20
17	0.65	0.66	0.19	0.42	0.61	0.19	0.21	0.29	0.20
18	0.64	0.65	0.18	0.42	0.60	0.19	0.21	0.29	0.19
29	0.62	0.64	0.17	0.42	0.59	0.19	0.21	0.28	0.18
20	0.62	0.64	0.17	0.42	0.59	0.19	0.21	0.28	0.18
21	0.61	0.62	0.14	0.39	0.58	0.18	0.20	0.28	0.16
22	0.60	0.61	0.16	0.39	0.56	0.18	0.20	0.28	0.17
23	0.60	0.61	0.14	0.39	0.56	0.18	0.20	0.26	0.16
24	0.59	0.60	0.16	0.39	0.55	0.18	0.20	0.26	0.17

Table 6.17 (b) Relative DOS Attackability Values (continued) derived from Table 6.16 (b).

Attack Session No.	***Relative DOS Attackability***								
	10	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***
1	5.72	5.80	5.63	3.83	5.69	5.76	1.85	5.12	4.33
2	2.75	2.82	2.65	1.96	2.71	2.78	0.95	2.35	2.10
3	2.06	2.14	1.97	1.32	2.03	2.10	0.64	1.69	1.58
4	1.60	1.67	1.50	1.00	1.56	1.63	0.48	1.15	1.23
5	1.13	1.20	1.20	0.82	1.26	1.16	0.38	0.89	0.88
6	1.15	1.22	1.06	0.71	1.12	1.19	0.34	0.70	0.90
7	1.02	1.09	0.91	0.64	0.97	1.06	0.26	0.72	0.80
8	0.90	0.97	0.80	0.58	0.86	0.94	0.28	0.60	0.71
9	0.83	0.90	0.73	0.54	0.79	0.86	0.25	0.52	0.66
10	0.76	0.83	0.66	0.49	0.72	0.79	0.23	0.43	0.60
11	0.72	0.79	0.62	0.47	0.68	0.76	0.22	0.43	0.58
12	0.68	0.76	0.59	0.44	0.65	0.72	0.20	0.37	0.55
13	0.66	0.73	0.56	0.42	0.62	0.70	0.19	0.34	0.53
14	0.62	0.70	0.53	0.40	0.59	0.66	0.18	0.29	0.50
15	0.61	0.68	0.52	0.38	0.58	0.65	0.17	0.25	0.50
16	0.59	0.66	0.49	0.36	0.55	0.62	0.17	0.26	0.48
17	0.60	0.67	0.50	0.36	0.56	0.64	0.17	0.26	0.49
18	0.59	0.66	0.49	0.36	0.55	0.62	0.17	0.25	0.48
29	0.58	0.65	0.48	0.35	0.54	0.61	0.17	0.24	0.47
20	0.58	0.65	0.48	0.35	0.54	0.61	0.17	0.24	0.47
21	0.56	0.64	0.47	0.35	0.53	0.60	0.16	0.22	0.46
22	0.55	0.62	0.46	0.35	0.52	0.59	0.16	0.23	0.45
23	0.55	0.62	0.46	0.34	0.52	0.59	0.16	0.22	0.45
24	0.54	0.61	0.44	0.34	0.50	0.58	0.16	0.23	0.44

6.5.3.2 Analysis of the Results

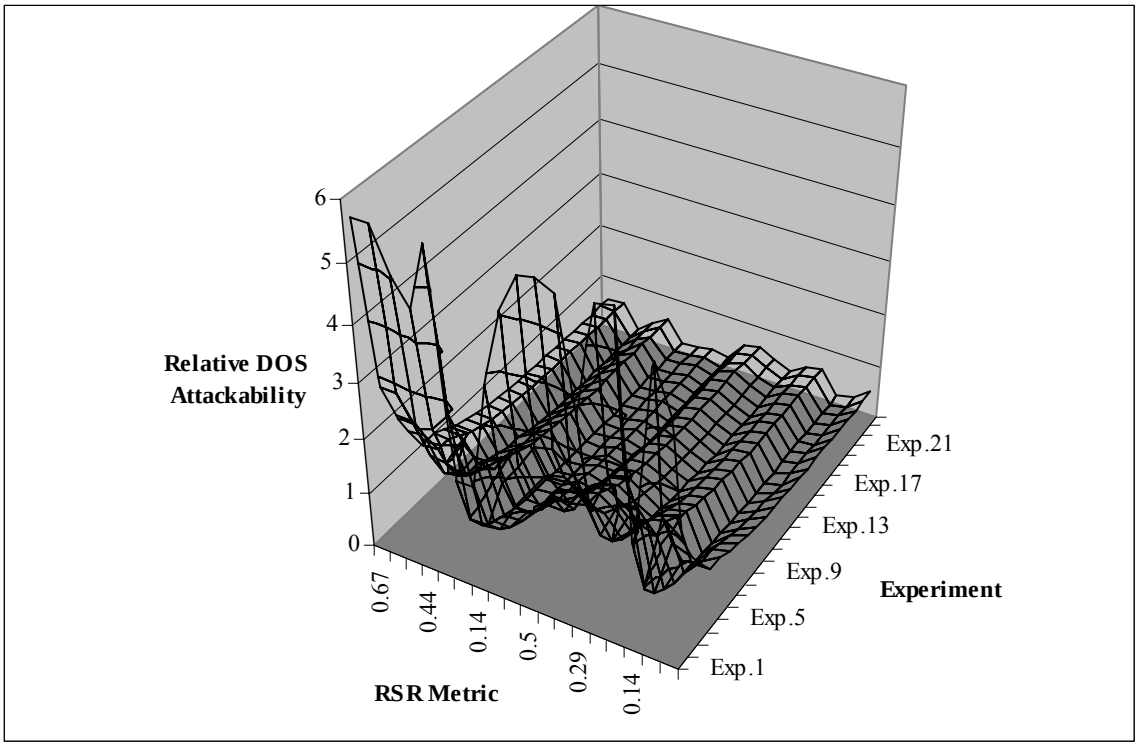

Figure 6.53. Plot Diagram for DOS

Inspection of the plot diagram of Figure 6.53 suggests that there seems to exist certain correlation between *RSR* Metric and relative DOS attackability. Consequently, like previously, we conduct further study by carrying out relevant statistics analyses to test the existence of the relationship. Specifically, considering that the measurement defined in this work are at the ordinal scales level, accordingly, we choose correlation analysis as the major analysis approach for the data collected in our experiments.

Table 6.18 presents the correlation coefficients between *RSR* metrics (provided in Table 6.14) and the relative DOS attackability metrics (presented in Table 6.17). Figure 6.54 presents the analysis results of the correlation coefficients based on data provided in Table 6.18. Note that the estimate of the mean of the correlation coefficients is around 0.62, which indicates a positive correlation between *RSR* metric and DOS attackability. The estimated value of the median is around 0.63, which also signals a correlation between the *RSR* metric and relative DOS attackability. From the histogram (see Figure

6.54) we observe that most of the correlation coefficients occur in the range 60-70% for this sample.

Table 6.18 Correlation Coefficients between RSR and Relative DOS Attackability

Experiment No.	*Correlation Coefficients*
1	0.645800
2	0.658882
3	0.663769
4	0.668361
5	0.652276
6	0.654074
7	0.633477
8	0.660126
9	0.650809
10	0.637843
11	0.642901
12	0.628492
13	0.619833
14	0.605254
15	0.592644
16	0.601441
17	0.601293
18	0.595712
19	0.591132
20	0.591132
21	0.581852
22	0.587568
23	0.583092
24	0.588955

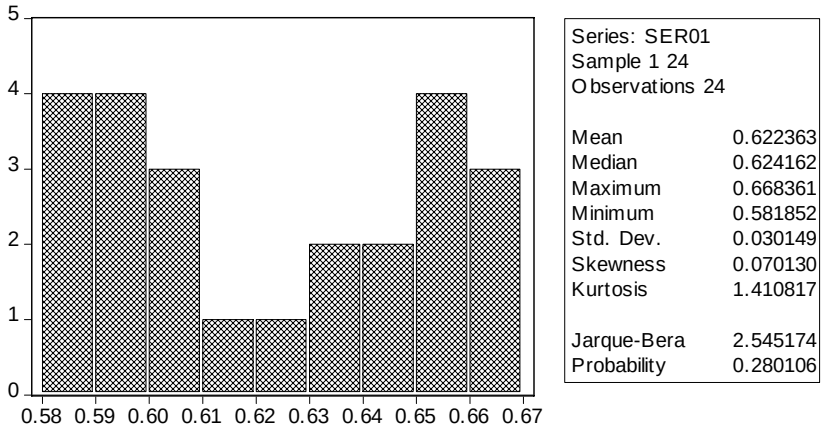

Figure 6.54. Correlation Analysis Results for DOS

Like previously, we conduct further test of the strength of the correlation between *RSR* Metric and relative DOS attackability through hypothesis testing and confidence interval analysis.

We begin with testing that the hypothesis of the average correlation between the *RSR* Metric and relative DOS attackability is always greater or equal to 0.60, which is considered to be strong correlation. We establish that the null hypothesis is a composite hypothesis and that the alternative hypothesis is one-sided:

$$H_0 : \textit{Average Correlation} \geq 0.60 \qquad \textit{versus} \qquad H_a : \textit{Average Correlation} < 0.60$$

We choose the confidence level to be 0.95, and assume the correlation follows a normal distribution. The appropriate test statistic has a t-distribution. With a sample size of 24, the critical value is -1.71.

Analysis of the collected correlations between *RSR* metric and relative DOS attackability (see Figure 6.54) results in the mean to be 0.62, and the standard deviation to be 0.03, which yields the computed test value to be 3.27. Since the calculated value lies in the acceptance region (3.27>-1.71), we conclude that the correlation between *RSR* Metric and relative DOS attackability is strong.

The result of the confidence analysis shows that 95% of time, the true value of the average correlation between the two variables will be covered by the confidence interval between 0.61 and 0.63 (0.61<correlation<0.63), and therefore we conclude that the correlation is strong.

Therefore we reach the same conclusion as in the previous study based on FS Application. We can conclude that at least under the current experimental conditions our hypothesis is supported.

6.6 Summary

In this chapter we have initiated the gathering of empirical evidences supporting the meaningfulness of our empirical security design hypotheses, which represent the backbone of the proposed quantitative security analysis framework. Using two different

web applications, we studied through the methodology of case studies the impact of some of the internal security-related attributes identified in this work on attackability of service-oriented software architecture. We also observed that at least two of the empirical validation hypotheses defined in Chapter 3 were supported by the empirical studies conducted in this chapter. Although we conducted these empirical studies in a restricted context, the empirical validation criteria defined in this chapter are applicable to other contexts and resources.

We recognize that some of the attackability data in this dissertation were collected without taking into consideration the impact of the some of the factors characterizing the experiment environment, such as the data collected for the reward and effort of application DOS attack. Firstly, we do not claim that our attackability measures at the current framework are definitive. We believe that more refined and representative attackability measures can be derived in the future as we explore deeper the research work on software attacks and intrusion analysis. In our current research, we focused on showing how diverse software attackability can be captured quantitatively and related with internal security metrics in our framework. Secondly, we have tried to minimize the environment impact in our attack experiments by conducting experiments using two different software applications under different time period and running repeated attack sessions for each software application. Based on the analysis results, we can make the statement that our hypothesis is supported at least under the current experimental conditions.

We also recognize that it is not sufficient to infer a general correlation between, for instance, the amount of coupling or complexity, and the likelihood of successful attack based on only two case studies. Although many empirical studies would be required to draw a general conclusion, the current study is a good step in this direction. The above empirical result is important for software security architects because it gives them a ground to make appropriate and effective decisions when designing and implementing secure software systems. It allows establishing better understanding of the underpinnings and intricacies of secure software and thereby leading to better software products.

Chapter 7

Concluding Remarks

In this section, we summarize our work and discuss future work.

7.1 Conclusions

The dramatic increase in the number of software attacks has put attackability in the forefront of quality concerns in the development of software systems. There is a growing consensus towards the use of software metrics as quality predictors. Metrics can be used to capture quantitatively important features of software systems and construct meaningful prediction models to guide the development of software products.

Security analysis at the architecture level is generally a difficult task. Traditionally, architectural level security analysis has relied solely on human expertise, with little room for automation. In this context, the level of expertise usually has a big impact on the analysis results. As a result, the outcome of the analysis may not be consistent if different security personnel are employed.

In order to increase the level of objectivity of architecture security analysis, it is necessary to identify and develop systematic guidelines and mechanisms that can be used both by experts and non-experts. In this work, we outlined the foundation of an automated framework for quantitative security analysis of service-oriented architecture. The framework involves two layers: metric definition and validation layer, and metric application layer. The metric definition layer consists of a set of internal software attributes which may impact directly software security, and a set of related measurement properties. The measurement properties can be used as basis for formal validation of corresponding metrics. The metric application layer involves a measurement abstraction, the USIE model, which may be used to compute security metrics from service-oriented

software architecture. The metric application layer also involves various security analysis techniques that may be used in conjunction of computed metrics to assess the risk exposure of the software architecture to specific kind of software attacks.

In order to gather empirical evidences supporting the proposed measurement framework, we have investigated through various case studies the attackability of an open source web-based application with respect to specific types of attacks. The methodology employed may be adapted and applied to many other applications and many other types of software attacks.

To conclude, we can say that in this work, we have tackled several important issues in the nascent field of quantitative security analysis at the software architecture level. Our main contribution is to have laid down foundation for both theoretical and empirical studies in this field. However, several important research challenges remain unsolved. We summarize those issues in the next section since we plan to tackle them in future work.

7.2 Future Work

Our future work will focus on further refining and extending the foundational work laid down in this dissertation by investigating many other types of software attacks and related internal software attributes and metrics. We also need to strengthen the empirical foundation of our framework by exploring stronger experimental methods beyond the methodology of case study. Formal experiments and surveys are extremely difficult to conduct in the area of software engineering. The diversity and rapid changes of software development techniques make it very difficult to obtain enough resources for formal experiments. Nonetheless, as mentioned earlier, it is not sufficient to infer a general correlation between specific internal software attribute and the likelihood of successful attack based on only one case study. So we need to investigate how formal experiments can be adapted and applied to the highly complex and evolving field of software security.

In this regard, one of the key issues to deal with is empirical evaluation data gathering. Unlike other fields of research, currently there is no standard or public dataset that may be used for empirical studies in quantitative architecture security analysis. Efforts are currently being undertaken through the Predictor Models In Software

Engineering (PROMISE) workshop series to develop a public repository of datasets provided by various research groups. The datasets presented in this work have been donated to the PROMISE repository and, as such, is publicly available to other researchers [49]. But these efforts are still at an early stage and largely remain confined at the level of the research community. For this kind of public dataset to be really effective, it should include field data collected from real development and operational environments. But due to privacy issues or for competitiveness or fear of lawsuits, companies are in general very reluctant to share this kind of data.

It would be necessary in future work to develop explicit attackability prediction models. Our current approach to attackability prediction is an indirect one, in the sense that it relies on the empirical security design hypotheses outlined in chapter 3. We base ourselves on these hypotheses to infer some relationships between attackability and internal software attributes, and use these relationships to guide security analysis process. It would be interesting to go beyond this implicit model by formalizing these relationships under the form of, for instance, some mathematical equations or formulas explicitly linking attackability metrics to internal software metrics. Using regression techniques, it might be possible to develop such models. But once again for such models to be viable, we would need to validate those empirically using field data, which as mentioned above has yet to be publicly available.

Another important issue that should be addressed in future work is the need to develop aggregate attackability prediction models. Our current approach is to develop attack specific models. Although this approach is convenient from research perspective it can be cumbersome in practical environments. In practice, there are many different kinds of software attack patterns, and new ones are being invented regularly. Developing a separate attackability model for each of those attack patterns is impossible. At this stage of our work, our recommendation for practitioners is to conduct proper risk analysis at the beginning of the design process, and prioritize the identified risks. Attackability analysis can then focus only on the most important risks in terms of assets and feasibility. Our goal, in future work, is to go beyond the current paradigm by developing aggregate attackability models which can be used to predict any kind of security attacks, known or

unknown. This might necessitate using machine learning techniques to predict general attacking behaviors and conditions.

We will also need in the future to extend the theoretical foundation of our work by identifying more security-related attributes and by addressing completeness issues. As indicated earlier, the formal system proposed in chapter 3 for theoretical validation of metrics provides only necessary conditions for validity. We need to establish the soundness of the proposed formal system by addressing completeness issue. This is a hard problem which has not yet been addressed in previous works on software measurement theory [8], [65], [72].

Finally, we also plan in the future to finalize the implementation of the STEM tool suite and have it adopted by the software community. This is an important aspect of our framework, because our ultimate goal in this work is to reduce through automation the level of dependence on expert knowledge in architecture security analysis.

Bibliography

[1] J. Alves-Foss, S. Barbosa, "Assessing Computer Security Vulnerability", *ACM SIGOPS Operating Systems Review Vol. 29, No. 3, p. 3-13*, 1995.

[2] M. Andrews, J. A. Whittaker*, How to Break Web Software*, Addison-Wesley, 2005.

[3] A. Arsanjani, "Service Oriented Modeling and Architecture", *IBM developer Works*, November 9th, 2004.

[4] S. Beattie, S. Arnold, C. Cowam, P. Wagle, C. Wright, A. Shostack, "Timing the Application of Security Patches for Optimal Uptime", *Proceedings of LISA'02, 16th Systems Administration Conference*, 2002.

[5] B. Blakley, C. Heath, *et al.*, "Security Design Patterns", *Technical Guide, Document Number: G031*, The Open Group, April 2004.

[6] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language Reference Manual*, Addison Wesley Longman, 1999.

[7] L. C. Briand, K. E. Eman, S. Morasca, "Theoretical and Empirical Validation of Software Product Measures", *International Software Engineering Research Network, technical report #ISERN-95-03*, 1995.

[8] L. C. Briand, S. Morasca, V. R. Basili, "Property-Based Software Engineering Measurement", *IEEE TSE, Vol. 22, No. 1*, Jan.1996.

[9] S. Brocklehurst, B. Littlewood, T. Olovsson, E. Johsson, "On Measurement of Operational Security", *Proceedings of the 9th Annual Conference on Computer Assurance*, June, 1994.

[10] H. Browne, J. McHugh, W. Arbaugh, W. Fithen, "A trend Analysis of Exploitations", *IEEE Symposium on Security and Privacy*, 2001.

[11] L. S. Cauman, *First-Order Logic*, Berlin: Walter de Gruyter, 1998.

[12] S. R. Chidamber, Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE TSE, Vol. 20, No. 6, pp. 476-493*, June 1994.

[13] A. Chou, J. Yang, B. Chelf, S. Hallen, D. Engler, "An empirical study of operating systems errors", ACM Symposium on Operating Systems Principles, pp. 73–88, Oct. 2001.

[14] CCITSE, "Common Criteria for Information Technology Security Evaluation", *published as [ISO/IEC 15408]*, 2005.

[15] M. Dacier, Y. Deswarte, "Privilege Graph: an Extension to the Typed Access Matrix Model", *Lecture Notes in Computer Science, 875:319--334*, 1994.

[16] Y. Deng, J. Wang, J. Tsai, and K. Beznosov, "An approach for modeling and analysis of security system architectures", *IEEE Trans. on Knowledge and Data Engineering, Vol. 15, No.5, pp. 1099-1119,* 2003.

[17] J. H. P. Eloff, "Selection Process for Security Packages", *Computers & Security, Vol. 2, No. 3*, 1983.

[18] M. Endrei, *Patterns: Service-oriented Architecture and Web Services*, Redbook, SG24-6303-00, April 2004.

[19] N. Fenton, *Software Metrics: A rigorous Approach*, *Chapman & Hall*, 1991.
[20] N. Fenton, "Software Measurement: A Necessary Scientific Basis", *IEEE TSE*, *Vol. 20, No. 3, pp. 199-206*, March 1994.
[21] N. Fenton, M. Meil, "A Critique of Software Defect Prediction Models", *IEEE TSC, Vol, 25, No. 3*, May/June, 1999.
[22] D. Frank, "Agencies Seek Security Measures", *CIO Magazine*, June 19, 2000.
[23] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D. Nassar, H. Ammar, A. Mili, "Architectural Level Risk Analysis using UML", *IEEE TSE, Vol. 29, No. 10,* October, 2003.
[24] S. Frank, W. Snyder. "Threat Modeling", *Redmond, WA: Microsoft Press*, 2004.
[25] M. Graw, "Testing for Security During Development: Why we should scrap penetrate-and patch", *IEEE Aerospace and Electronic Systems, Vol. 13, No. 4, pp. 13-15*, April, 1998.
[26] J. Gray, "A census of tandem system availability between 1985 and 1990"*, IEEE TSE, vol. 39, No. 4*, Oct. 1990.
[27] B. Henderson-Sellers, *Object-Oriented Metrics*, Prentice Hall, USA, 1996.
[28] M. Howard, J. Pincus, J. M. Wing, "Measuring Relative Attack Surfaces", *Proceeding of Workshop on Advanced Developments in Software and System Security*, December, 2003.
[29] M. Howard, S. Lipner, "The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software", *Microsoft Press*, 2006.
[30] ITSEC, "Information Technology Security Evaluation Criteria - Harmonized Criteria of France Germany, the Netherlands, and the United Kingdom", *Version 1.1, Published by Dept. of Trade and Industry*, London, 1991.
[31] J. Gonzalez Nieto, K. Viswanathan, C. Boyd, A. J. Clark and E. P. Dawson, "Key Recovery for the Commercial Environment", *International Journal of Information Security (IJIS)*, 1(3):161--174, November 2002.
[32] J. Jacob, "Basic Theorems About Security", *Journal of Computer Security, 1(4), pp 385-411*, 1992.
[33] G. Jelen, "SSE-CMM Security Metrics", *NIST and CSSPAB Workshop*, Washington, D.C., 13-14 June, 2000.
[34] S. Johnston, "Modeling Security Concerns in Service-Oriented Architecture"*, IBM Developer Works*, June 25th, 2004.
[35] J. Jürjens, "Principles for Secure Systems Design", *PhD thesis*, Oxford University Computing Laboratory, Trinity Term, 2002.
[36] J. Jürjens, "Model-based Security Engineering with UML", *FOSAD 2004/05 Tutorial volume, LNCS, © Springer Verlag*, 2005.
[37] R. Kazman, G. Abowd, L. B. Clements, "Scenario-based analysis of software architecture", *Proc. Of IEEE, Vol. 13, Issue 6, pp. 47 – 55*, November, 1996.
[38] R. A. Kemmerer, "Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels", *ACM Transactions on Computer Systems, 1:3, pp. 256-277*, August 1983.
[39] B. Kitchenham, S. L. Pfleeger, N. Fenton, "Towards a Framework for Software Measurement Validation", *IEEE TSE, vol.21, No. 12, pp. 929-943*, December, 1995.

[40] B. Kitchenham, S. L. Pfleege, N. Fenton, "Reply to: Comments on Towards a framework for Software Measurement Validation", *IEEE TSE, Vol. 23, No. 3, pp. 189*, March, 1997.

[41] I. Lee, R. Iyer, "Faults, symptoms, and software fault tolerance in the tandem GUARDIAN operating system", *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1993.

[42] M. Y. Liu, I. Traore, "UML-based Security Measures of Software Products", *International Workshop on Methodologies for Pervasive and Embedded Software (MOMPES'04)*, Hamilton, Ontario, Canada, June, 2004.

[43] M. Y. Liu, I. Traore, "Measurement Framework for Software Privilege Protection based on User Interaction Analysis", *11th IEEE International Software Metrics Symposium,* Como, Italy, September 19-22 , 2005.

[44] M. Y. Liu, I. Traore, "Empirical Relationships between attackability and coupling: case study for DOS", *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, Ottawa, Canada, June 10, 2006, pages 57-64.

[45] D. Ghindici, M. Y. Liu, G. Grimaud, I. Ryl, I. Traore, "Integrated Security Verification and Validation: Case Study", *Proceedings of 2nd IEEE LCN Workshop on Network Security*, Tampa, Florida, U.S.A., November 14, 2006.

[46] M.Y. Liu, I. Traore, "Quantitative Security Analysis for Service Oriented Software Architecture", *Technical Report No. ECE-07-5*, University of Victoria, ECE Department, Victoria, BC, Canada, 2007.

[47] M. Y. Liu, I. Traore, "Complexity Measures for Secure Service-Oriented Software Architectures", in the Proceedings of the 3rd IEEE International Predictor Models in Software Engineering (PROMISE) Workshop, May 20, 2007, Minneapolis, Minnesota, USA, in conjunction with 29th International Conference on Software Engineering (ICSE); (10 pages).

[48] M.Y. Liu, I. Traore, "Systematic Security Analysis for Service-Oriented Software Architectures", in the Proceedings of 3rd IEEE International Workshop on Service-Oriented System Engineering (SOSE07), Hong-Kong, China, Oct. 24-26, 2007 (10 pages).

[49] M. Y. Liu, I. Traore, "Properties for Security Measures of Software Products", *Journal of Applied Mathematics & Information Sciences*, 1 (2), May 2007, pages 129-156.

[50] M.Y. Liu, I. Traore, "Software Product Metrics for Security Analysis of Service-Oriented Architectures", *submitted to the International Journal of Empirical Software Engineering*, September 2007, (37 pages).

[51] P. Manadhata, J. M. Wing, "Measuring a System's Attack Surface", *CMU-CS-04-102 Technical Report*, January 2004.

[52] A. C. Melton, D. A. Gustafson, J. M. Bieman, A. L. Baker, "A mathematical perspective for software measures research," *Software Eng. J., vol. 5, no. 5, pp. 246-254*, September, 1990.

[53] D. A. Menasce, V. A . F. Almeida, *Capacity Planning for Web Services*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[54] Meyer, *Object-Oriented Software Construction*, *2nd Edition, Prentice-Hall,* 1997.

[55] J. K. Millen, "Survivability Measure", *Research Report*, *Computer Science Laboratory (CSL)*, SRI International, Menlo Park, CA, USA, 2000.

[56] S. Morasca, L. C. Briand, V. R. Basili, E. J. Weyuker, M. V. Zelkowitz, "Comments on Towards a framework for Software Measurement Validation", *IEEE TSE, Vol. 23, No. 3, pp. 187-188*, March 1997.

[57] S. Morasca, L. C. Briand, "Towards A Theoretical Framework For Measuring Software Attributes", *Fourth International Software Metrics Symposium, p. 119*, 1997.

[58] D. M. Nicol, W. H. Sanders, K. S. Trivedi, "Model-Based Evaluation: from dependability to Security", *IEEE TDSC, Vol. 1, No. 1, pp. 48-65,* January-March 2004.

[59] F. Nielsen, "Approaches to Security Metrics", *NIST and CSSPAB Workshop*, Washington, D.C., 13-14 June, 2000.

[60] J. Saltzer, M. Schroeder, "The Protection of Information in Computer Systems", *Proc. Of IEEE, Vol. 63, No. 6, pp. 1278-1308,* September, 1975.

[61] R. S. Sandhu, "The Typed Access Matrix Model", *Proc 1992 IEEE Symposium on research in Security and Privacy, pp. 122-136*, May 4-6, 1992.

[62] T. A. Smith, "User Definable Domains as a Mechanism for Implementing the Least Privilege Principle", *Proceedings of 9th National Computer Security Conference, pp. 143-148,* September, 1986.

[63] L. Smith, "Cryptographic Algorithm Metrics", *NIST and CSSPAB Workshop*, Washington, D.C., June, 2000.

[64] M. Sullivan, R. Chillarge, "Software defects and their impact on system 118 availability"*, Proceedings of the International Symposium on Fault-Tolerant Computing*, June, 1991.

[65] J. Tian, M. V. Zelkowitz, "A formal program complexity model and its applications", *J. Syst. Soft.*, *vol. 17, pp. 253-266,* 1992.

[66] I. Traore, "An Outline of PVS Semantics for UML Statecharts", *Journal of Universal Computer Science (JUCS)*, *Springer Pub. Co., Vol. 6, No. 11, pp. 1088-1108*, November, 2000.

[67] R. B. Vaughn, "Are Measures and Metrics for Trusted Information Systems Possible?", *Workshop on Information Security System Scoring and Ranking*, Williamsburg, VA, USA, May 21-23, 2001.

[68] J. Viega, G. McGraw, *Building Secure Software*, Addison-Wesley, 2001.

[69] J. Voas, A. Ghosh, G. McGraw, F. Charron, K. Miller, "Defining an Adaptive Software Security Metric from a Dynamic Software Failure Tolerance Measure", *Proceedings of the 11th Annual Conference on Computer Assurance*, 1996.

[70] J. Wang, X. He, Y. Deng, "Introducing Software Architecture Specification and Analysis in SAM through an Example", *Information and Software Technology, vol. 41, no. 7, pp. 451-467*, 1999.

[71] C. Wang, W. A. Wulf, "Towards a framework of security measurement", *20th NISSC Proceedings*, Baltimore, Maryland, October, 1997.

[72] E. J. Weyuker, "Evaluating Software Complexity Measures", *IEEE TSE, vol. 14, no. 9, pp. 1357-1365*, Sept. 1988.

[73] J. A. Whittaker, H. H. Thompson, *How to Break Software Security*, Pearson Education, 2004.

[74] D. Xu, K. Nygard, "A Threat-Driven Approach to Modeling and Verifying Secure Software", *ASE'05*, Long Beach, California, USA, November 7–11, 2005.

[75] O. Zimmermann, P. Krogdahl, C. Gee, "Elements of Service-Oriented Analysis and Design", *IBM developer Works*, June 2nd, 2004.

[76] C. Koten 1 and A.R. Gray, "An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability", *The Information Science Discussion Paper Series*, March 2005.

[77] C. Wohlin, L. Lundberg and M. Mattsson, "Trade-off Analysis of Software Quality Attributes", *Software Quality Journal*, *pp 327-328*, November 14, 2005.

[78] Mary J. Allen, Wendy M. Yan, *Introduction to Measurement Theory*, Waveland Press; 1 edition, 2001.

[79] S. Lafrance, J. Mullins, "Using admissible interference to detect denial of service vulnerabilities", *Sixth International Workshop in Formal Methods (IWFM'03)*, Dublin, 2003.

[80] Thomas R. Peltier, *Information Security Risk Analysis*, second edition, CRC Press LLC, 2005.

[81] Moore David, *Basic Practice of Statistics*, fourth edition, WH Freeman Company, 2006.

Appendix A: Theoretical Validation for Sample Metrics from Existing Measurement Frameworks

We present in this section, the detailed proofs for the verification of the sample metrics used in chapter 3 to illustrate our formal validation framework.

Metrics *attack_surf$_1$*

$$attack\_surf_1(\bar{s}) = \sum_{a \in S_{Attack}} count(a \bullet \bar{s}) \times payoff(a)$$

$$attack\_surf_1(s) = \begin{cases} \sum_{a \in S_{Attack}} count(\bigcup_{t \in \underline{s}} (a \bullet \bar{t})) \times payoff(a) & (\text{if } \underline{s} \neq \varnothing) \\ attack\_surf_1(\bar{s}) & (\text{if } \underline{s} = \varnothing) \end{cases}$$

For *attack_surf$_1$* to be considered as a service complexity metric, it should satisfy axioms AC1~AC5.

1) Axiom AC1: $\forall \bar{s} \in \bar{S}\ Complexity(\bar{s}) \geq 0$

$\forall \bar{s} \in \bar{S}\ \forall a \in S_{attack}\ count(a \bullet \bar{s}) \geq 0, payoff(s_j) \in [0,1] \Rightarrow$
$attack\_surf_1(\bar{s}) = \sum_{a \in S_{Attack}} count(a \bullet \bar{s}) \times payoff(a) \geq 0.$ □

Therefore, the *attack_surf$_1$* metric satisfies axiom AC1.

2) Axiom AC2: $\forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ Complexity(\bar{s}) \geq Complexity(t)$

*Postulate P*11: $\forall a \in S_{attack}\ \forall s \in S\ \forall \bar{t} \in \underline{s}\ [(a \bullet \bar{t}) \subseteq (a \bullet s)] \wedge [(a \bullet s) \subseteq (a \bullet \bar{s})]$

$$Postulate\ P11 \Rightarrow \forall s \in S \left[\bigcup_{\bar{t}\in\underline{s}}(a \bullet \bar{t})\right] \subseteq (a \bullet \bar{s}) \Rightarrow \forall a \in S_{Attack}\ count(\left[\bigcup_{\bar{t}\in\underline{s}}(a \bullet \bar{t})\right]) \le count(a \bullet \bar{s})$$

$$\Rightarrow \forall a \in S_{Attack}\ count(\left[\bigcup_{\bar{t}\in\underline{s}}(a \bullet \bar{t})\right]) \times payoff(a) \le count(a \bullet \bar{s}) \times payoff(a)$$

$$\Rightarrow \sum_{a\in S_{attack}} count(\left[\bigcup_{\bar{t}\in\underline{s}}(a \bullet \bar{t})\right]) \times payoff(a) \le \sum_{a\in S_{attack}} count(a \bullet \bar{s}) \times payoff(a)$$

$$\Rightarrow \forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ attack\_surf_1(t) \le attack\_surf_1(\bar{s}). \square$$

Therefore, the *attack_surf1* metric satisfies axiom AC2.

3) Axiom AC3: $\forall s \in S\ \forall \bar{t} \in \underline{s}\ Complexity(\bar{t}) \le Complexity(s)$

$$\forall s \in S\ \underline{s} \neq \varnothing\ \forall \bar{t} \in \underline{s}\ (a \bullet \bar{t}) \subseteq \bigcup_{\bar{t}\in\underline{s}}(a \bullet \bar{t})$$

$$\Rightarrow attack\_surf_1(s) = \sum_{a\in S_{Attack}} count(\bigcup_{\bar{t}\in\underline{s}}(a \bullet \bar{t})) \times payoff(a)$$

$$\ge \sum_{a\in S_{Attack}} count(a \bullet \bar{t}) \times payoff(a) \quad (\bar{t} \in \underline{s})$$

$$= attack\_surface_1(\bar{t}) \quad (\bar{t} \in \underline{s}). \square$$

Therefore, the *attack_surf1* metric satisfies axiom AC3.

4) Axiom AC4: $\forall s \in S\ \underline{s} = \varnothing \Rightarrow Complexity(\bar{s}) = Complexity(s)$

By definition, $\underline{s} = \varnothing \Rightarrow attack\_surf_1(s) = attack\_surf_1(\bar{s}). \square$, therefore, the *attack_surf1* metric satisfies axiom AC4.

5) Axiom AC5: $\forall \bar{s}, \bar{t} \in \bar{S}\ \bar{s} \subseteq \bar{t} \Rightarrow Complexity(\bar{s}) \le Complexity(\bar{t})$

$$\forall \bar{s}, \bar{t} \in \bar{S}\ \bar{s} \subseteq \bar{t}$$

$$\Rightarrow \forall a \in S_{attack}\ (a \bullet \bar{s}) \subseteq (a \bullet \bar{t})$$

$$\Rightarrow \forall a \in S_{attack}\ count(a \bullet \bar{s}) \le count(a \bullet \bar{t})$$

$$\Rightarrow \sum_{a\in S_{Attack}} count(a \bullet \bar{s}) \times payoff(a) \le \sum_{a\in S_{Attack}} count(a \bullet \bar{t}) \times payoff(a)$$

$$\Rightarrow attack\_surf_1(\bar{s}) \le attack\_surf_1(\bar{t}). \square$$

Therefore, the *attack_surf1* metric satisfies axiom AC5.

In summary, the *attack_surf1* satisfies axioms AC1~AC5, therefore, can be considered as a service complexity metric in our framework.

Metrics *attack_surf2*

$$attack\_surface_2(\overline{s_t},\overline{s_u}) = cardinality(resource(\overline{s_t}) \cap resource(\overline{s_u}))$$

$$attack\_surface_2(s_t,s_u) = \begin{cases} cardinality(\bigcup_{t_i \in \underline{s_t}} resource(\overline{t_i}) \cap \bigcup_{t_j \in \underline{s_u}} resource(\overline{t_j})) & (\text{if } \underline{s_t} \neq \varnothing \wedge \underline{s_u} \neq \varnothing) \\ cardinality(\bigcup_{t_i \in \underline{s_t}} resource(\overline{t_i}) \cap resource(\overline{s_u})) & (\text{if } \underline{s_t} \neq \varnothing \wedge \overline{s_u} = \varnothing) \\ cardinality(resource(\overline{s_t}) \cap \bigcup_{t_j \in \underline{s_u}} resource(\overline{t_j})) & (\text{if } \underline{s_t} = \varnothing \wedge \overline{s_u} \neq \varnothing) \\ attack\_surface_2(\overline{s_t},\overline{s_u}) & (\text{if } \underline{s_t} = \varnothing \wedge \underline{s_u} = \varnothing) \end{cases}$$

For *attack_surf₂* to be considered as a service coupling metric, it should satisfy axioms AD1~AD8.

1) Axiom AD1: $\forall \overline{s_1},\overline{s_2} \in \overline{S}\ Coupling(\overline{s_1},\overline{s_2}) \geq 0$

$attack\_surface_2(\overline{s_1},\overline{s_2}) = cardinality(resource(\overline{s_1}) \cap resource(\overline{s_2})) \geq 0.$ □

Therefore, the *attack_surf₂* metric satisfies axiom AD1.

2) Axiom AD2: $\forall \overline{s_1},\overline{s_2} \in \overline{S}\ \overline{s_1} \propto \overline{s_2} = \varnothing \Rightarrow Coupling(\overline{s_1},\overline{s_2}) = 0$

$\forall \overline{s_1},\overline{s_2} \in \overline{S}, \overline{s_1} \propto \overline{s_2} = \varnothing$

$\Rightarrow resource(\overline{s_1}) \cap resource(\overline{s_2}) = \varnothing$

$\Rightarrow attack\_surface_2(\overline{s_1},\overline{s_2}) = cardinality(resource(\overline{s_1}) \cap resource(\overline{s_2})) = 0.$ □

Therefore, the *attack_surf₂* metric satisfies axiom AD2.

3) Axiom AD3: $\forall \overline{s_1},\overline{s_2} \in \overline{S}\ \forall t_1 \in \overline{s_1}\ \forall t_2 \in \overline{s_2} \Rightarrow Coupling(\overline{s_1},\overline{s_2}) \geq Coupling(t_1,t_2)$

*Postulate P*21: $\forall \overline{s} \in \overline{S}\ resource(\overline{s}) = \bigcup_{t \in \overline{s}} resource(t)$

*Postulate P*21 $\Rightarrow \forall \overline{s_1},\overline{s_2} \in \overline{S}, \forall t_1 \in \overline{s_1}\ \forall t_2 \in \overline{s_2}\ resource(t_1) \subseteq resource(\overline{s_1}) \wedge resource(t_2) \subseteq resource(\overline{s_2})$

$\Rightarrow resource(t_1) \cap resource(t_2) \subseteq resource(\overline{s_1}) \cap resource(\overline{s_2})$

$\Rightarrow cardinality(resource(t_1) \cap resource(t_2)) \leq cardinality(resource(\overline{s_1}) \cap resource(\overline{s_2}))$

$\Rightarrow attack\_surface_2(t_1,t_2) \leq attack\_surface_2(\overline{s_1},\overline{s_2}).$ □

Therefore, the *attack_surf₂* metric satisfies axiom AD3.

4) Axiom AD4: $\forall t_1,t_2 \in S\ \forall \overline{s_i} \in \underline{t_1}\ \forall \overline{s_j} \in \underline{t_2}\ Coupling(t_1,t_2) \geq Coupling(\overline{s_i},\overline{s_j})$

*Postulate P*22: $\forall s \in S\ resource(s) = \bigcup_{\overline{t} \in \underline{s}} resource(\overline{t})$

*Postulate P*22

$\Rightarrow \forall t_1, t_2 \in S\ \forall \bar{s}_i \in \underline{t_1}, \bar{s}_j \in \underline{t_2}, resource(\bar{s}_i) \subseteq resource(t_1), resource(\bar{s}_j) \subseteq resource(t_2)$

$\Rightarrow resource(\bar{s}_i) \cap resource(\bar{s}_j) \subseteq resource(t_1) \cap resource(t_2)$

$\Rightarrow cardinality(resource(\bar{s}_i) \cap resource(\bar{s}_j)) \le cardinality(resource(t_1) \cap resource(t_2))$

$\Rightarrow attack\_surface_2(\bar{\bar{s}}_i, \bar{\bar{s}}_j) \le attack\_surface_2(t_1, t_2).$ □

Therefore, the *attack_surf*$_2$ metric satisfies axiom AD4.

5) Axiom AD5: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) \le \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} Coupling(\bar{\bar{s}}_i, \bar{\bar{s}}_j)$

*Postulate P*22: $\forall s \in S\ resource(s) = \bigcup_{\bar{t} \in \underline{s}} resource(\bar{t})$

1) *Postulate P*22 $\Rightarrow \forall s_1, s_2 \in S\ attack\_surface_2(s_1, s_2) = resource(s_1) \cap resource(s_2)$

2) *Postulate P*22 $\Rightarrow \forall s_1, s_2 \in S\ resource(s_1) \cap resource(s_2) = \left[\bigcup_{\bar{s}_i \in \underline{s_1}} resource(\bar{s}_i)\right] \cap \left[\bigcup_{\bar{s}_j \in \underline{s_2}} resource(\bar{s}_j)\right]$

$\Rightarrow resource(s_1) \cap resource(s_2) = \bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} \left[resource(\bar{\bar{s}}_i) \cap resource(\bar{\bar{s}}_j)\right]$

$\Rightarrow cardinality(resource(s_1) \cap resource(s_2)) = cardinality\left(\bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} \left[resource(\bar{\bar{s}}_i) \cap resource(\bar{\bar{s}}_j)\right]\right)$

$\Rightarrow cardinality(resource(s_1) \cap resource(s_2)) \le \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} cardinality(resource(\bar{s}_i) \cap resource(\bar{s}_j))$

1) *and* 2) $\Rightarrow \forall s_1, s_2 \in S\ attack\_surface_2(s_1, s_2) \le \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} attack\_surface_2(\bar{\bar{s}}_i, \bar{\bar{s}}_j).$ □

Therefore, the *attack_surf*$_2$ metric satisfies axiom AD5.

6) Axiom AD6: $\forall \bar{\bar{s}}_1, \bar{\bar{s}}_2, \bar{s} \in \bar{S}\ \bar{\bar{s}}_1 \subseteq \bar{\bar{s}}_2 \Rightarrow Coupling(\bar{\bar{s}}_1, \bar{s}) \le Coupling(\bar{\bar{s}}_2, \bar{s})$

$\forall \bar{\bar{s}}_1, \bar{\bar{s}}_2, \bar{s} \in \bar{S}\ \bar{\bar{s}}_1 \subseteq \bar{\bar{s}}_2$

$\Rightarrow resource(\bar{\bar{s}}_1) \subseteq resource(\bar{\bar{s}}_2)$

$\Rightarrow resource(\bar{\bar{s}}_1) \cap resource(\bar{s}) \subseteq resource(\bar{\bar{s}}_2) \cap resource(\bar{s})$

$\Rightarrow cardinality(resource(\bar{\bar{s}}_1) \cap resource(\bar{s})) \le cardinality(resource(\bar{\bar{s}}_2) \cap resource(\bar{s}))$

$\Rightarrow attack\_surface_2(\bar{\bar{s}}_1, \bar{s}) \le attack\_surface_2(\bar{\bar{s}}_2, \bar{s}).$ □

Therefore, the *attack_surf*$_2$ metric satisfies axiom AD6.

7) Axiom AD7: $\forall \bar{\bar{s}}_1, \bar{\bar{s}}_2 \in \bar{S}\ Coupling(\bar{\bar{s}}_1, \bar{\bar{s}}_2) = Coupling(\bar{\bar{s}}_2, \bar{\bar{s}}_1)$

$\forall \bar{\bar{s}}_1, \bar{\bar{s}}_2 \in \bar{S}\ resource(\bar{\bar{s}}_1) \cap resource(\bar{\bar{s}}_2) = resource(\bar{\bar{s}}_2) \cap resource(\bar{\bar{s}}_1)$

$\Rightarrow cardinality(resource(\bar{\bar{s}}_1) \cap resource(\bar{\bar{s}}_2)) = cardinality(resource(\bar{\bar{s}}_2) \cap resource(\bar{\bar{s}}_1))$

$\Rightarrow attack\_surface_2(\bar{\bar{s}}_1, \bar{\bar{s}}_2) = attack\_surface_2(\bar{\bar{s}}_2, \bar{\bar{s}}_1).$ □

Therefore, the *attack_surf*$_2$ metric satisfies axiom AD7.

8) Axiom AD8: $\forall s_1, s_2 \in S \; Coupling(s_1, s_2) = Coupling(s_2, s_1)$

$$\forall s_1, s_2 \in S$$

$$attack\_surface_2(s_1, s_2)$$

$$= \begin{cases} cardinality(\bigcup_{t_i \in \underline{s_1}} resource(\bar{t_i}) \cap \bigcup_{t_j \in \underline{s_2}} resource(\bar{t_j})) & (\text{if } \underline{s_1} \neq \varnothing \wedge \underline{s_2} \neq \varnothing) \\ cardinality(\bigcup_{t_i \in \underline{s_1}} resource(\bar{t_i}) \cap resource(\bar{s_2})) & (\text{if } \underline{s_1} \neq \varnothing \wedge \underline{s_2} = \varnothing) \\ cardinality(resource(\bar{s_1}) \cap \bigcup_{t_j \in \underline{s_2}} resource(\bar{t_j})) & (\text{if } \underline{s_1} = \varnothing \wedge \underline{s_2} \neq \varnothing) \\ attack\_surface_2(\bar{s_1}, \bar{s_2}) & (\text{if } \underline{s_1} = \varnothing \wedge \underline{s_2} = \varnothing) \end{cases}$$

$$= \begin{cases} cardinality(\bigcup_{t_i \in \underline{s_2}} resource(\bar{t_i}) \cap \bigcup_{t_j \in \underline{s_1}} resource(\bar{t_j})) & (\text{if } \underline{s_2} \neq \varnothing \wedge \underline{s_1} \neq \varnothing) \\ cardinality(\bigcup_{t_i \in \underline{s_2}} resource(\bar{t_i}) \cap resource(\bar{s_1})) & (\text{if } \underline{s_2} \neq \varnothing \wedge \underline{s_1} = \varnothing) \\ cardinality(resource(\bar{s_2}) \cap \bigcup_{t_j \in \underline{s_1}} resource(\bar{t_j})) & (\text{if } \underline{s_2} = \varnothing \wedge \underline{s_1} \neq \varnothing) \\ attack\_surface_2(\bar{s_2}, \bar{s_1}) & (\text{if } \underline{s_2} = \varnothing \wedge \underline{s_1} = \varnothing) \end{cases}$$

$$= attack\_surface_2(s_2, s_1) \square$$

Therefore, the *attack_surf₂* metric satisfies axiom AD8.

In summary, the *attack_surf₂* satisfies axioms AD1~AD8, therefore, can be considered as a service coupling metric in our framework.

Metrics *attack_surf₃*

$$attack\_surface_3(\bar{s})$$

$$= cardinality(\bigcup_{u \in \Pr inciple(\bar{s})} \{ \bigcup_{res_i \in Resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s}) \})$$

$$attack\_surface_3(s)$$

$$= \begin{cases} attack\_surface_3(\bar{s}) & (\text{if } \underline{s} = \varnothing) \\ cardinality(\bigcup_{t \in \underline{s}} \bigcup_{u \in \Pr inciple(\bar{t})} \{ \bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t}) \}) & (\text{if } \underline{s} \neq \varnothing) \end{cases}$$

For *attack_surf₃* to be considered as an excess privilege metric, it should satisfy axioms AP1~AP5.

1) Axiom AP1: $\forall \bar{s} \in \bar{S} \; EP(\bar{s}) \geq 0$

$$\forall \bar{s} \in \bar{S},\ attack\_surface_3(\bar{s}) = cardinality(\bigcup_{u \in \Pr inciple(\bar{s})} \{\bigcup_{res_i \in Resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s})\}) \geq 0. \square$$

Therefore, the *attack_surf3* metric satisfies axiom AP1.

2) Axiom AP2: $\forall s \in S\ \forall \bar{t} \in \underline{s}\ EP(s) \geq EP(\bar{t})$

$$\forall s \in S\ \forall \bar{t} \in \underline{s} \bigcup_{u \in \Pr inciple(\bar{t})} \{\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})\}$$

$$\subseteq \bigcup_{\bar{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\bar{t})} \{\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})\}$$

$$\Rightarrow \forall s \in S\ \forall \bar{t} \in \underline{s}\ cardinality(\bigcup_{u \in \Pr inciple(\bar{t})} \{\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})\})$$

$$\leq cardinality(\bigcup_{\bar{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\bar{t})} \{\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})\})$$

$$\Rightarrow \forall s \in S\ \forall \bar{t} \in \underline{s}\ attack\_surface_3(\bar{t}) \leq attack\_surface_3(s). \square$$

Therefore, the *attack_surf3* metric satisfies axiom AP2.

3) Axiom AP3: $\forall s \in S\ \underline{s} \neq \varnothing \Rightarrow EP(s) \leq \sum_{\bar{t}_i \in \underline{s}} EP(\bar{t}_i)$

$$\forall s \in S\ \underline{s} \neq \varnothing$$

$$\Rightarrow attack\_surface_3(s)$$

$$= cardinality(\bigcup_{\bar{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\bar{t})} \{\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})\})$$

$$\leq \sum_{\bar{t} \in \underline{s}} cardinality(\bigcup_{u \in \Pr inciple(\bar{t})} \{\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})\})$$

$$= \sum_{\bar{t} \in \underline{s}} attack\_surface_3(\bar{t}). \square$$

Therefore, the *attack_surf3* metric satisfies axiom AP3.

4) Axiom AP4: $\forall s \in S\ EP(\bar{s}) \geq EP(s)$

*Postulate P*31:

$$\forall s \in S\ \forall \bar{t} \in \underline{s}\ \forall u \in \Pr inciple(\bar{t})$$

$$\{\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t})\} \subseteq \{\bigcup_{res_i \in Resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s})\}$$

$$\left.\begin{array}{l}
1)\ \forall s \in S\ \underline{s} = \varnothing \Rightarrow attack_surface_3(s) = attack_surface_3(\bar{s}) \qquad (by\ definition) \\
2)\ \forall s \in S\ \underline{s} \neq \varnothing \Rightarrow \bigcup_{\bar{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\bar{t})} \left(\bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t}) \right) \\
\qquad \subseteq \bigcup_{u \in \Pr inciple(\bar{s})} \left(\bigcup_{res_i \in Resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s}) \right) \quad (P31) \\
\Rightarrow cardinality\left(\bigcup_{\bar{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\bar{t})} \{ \bigcup_{res_i \in Resource(\bar{t})} u \times res_i \times right(u, res_i) - LP(u, \bar{t}) \} \right) \\
\qquad \leq cardinality\left(\bigcup_{u \in \Pr inciple(\bar{s})} \{ \bigcup_{res_i \in Resource(\bar{s})} u \times res_i \times right(u, res_i) - LP(u, \bar{s}) \} \right) \\
\Rightarrow attack_surface_3(s) \leq attack_surface_3(\bar{s})
\end{array}\right\}$$

$\Rightarrow \forall s \in S\ attack\_surface_3(s) \leq attack\_surface_3(\bar{s}).$ □

Therefore, the *attack_surf₃* metric satisfies axiom AP4.

5) Axiom AP5: $\forall s \in S\ \underline{s} = \varnothing \Rightarrow EP(\bar{s}) = EP(s)$

By definition, $\forall s \in S\ \underline{s} = \varnothing \Rightarrow attack\_surface_3(s) = attack\_surface_3(\bar{s}).$ □, therefore, satisfies axiom AP5.

In summary, the *attack_surf₃* satisfies axioms AP1~AP5, therefore, can be considered as an excess privilege metric in our framework.

Metrics *MinMTTF*

$$MinMTTF(\bar{s}) = \underset{q \in Q_{\bar{s}}}{Min}\{MTTF(I_q)\}$$

$$MinMTTF(s) = \begin{cases} \sum_{\bar{t}_i \in \underline{s}} MinMTTF(\bar{t}_i) & (\text{if } \underline{s} \neq \varnothing) \\ MinMTTF(\bar{s}) & (\text{if } \underline{s} = \varnothing) \end{cases}$$

For *MinMTTF* to be considered as a mechanism strength metric, it should satisfy axioms AS1~AS7.

1) Axiom AS1: $\forall \bar{s} \in \bar{S}\ \ MStrength(\bar{s}) \geq 0$

$\forall \bar{s} \in \bar{S}\ \forall q \in Q_{\bar{s}}\ MTTF(I_q) \geq 0 \Rightarrow MinMTTF(\bar{s}) = \underset{q \in Q_{\bar{s}}}{Min}\{MTTF(I_q)\} \geq 0.$ □

Therefore, the *attack_surf₄* metric satisfies axiom AS1.

2) Axiom AS2: $\forall s \in S \ \underline{s} = \varnothing \Rightarrow MStrength(\bar{s}) = MStrength(s)$

By definition, $\underline{s} = \varnothing \Rightarrow MinMTTF(s) = MinMTTF(\bar{s})$. □ Therefore, the *attack_surf*$_4$ metric satisfies axiom AS2.

3) Axiom AS3: $\forall s \in S \ [\bar{s} \in (\overline{S} - \Gamma) \wedge \underline{s} = \varnothing] \ \Rightarrow MStrength(\bar{s}) = 0$

Note that by definition $MTTF(G_q) = t_{G_q} + \sum_{k \in out(G_q)} P_{G_q k} \times MTTF(k)$, and since G_q is the goal state, $out(G_q) = \varnothing$ and $t_{G_q} = 0$; this implies that $MTTF(G_q) = 0$.

$$
\begin{aligned}
&\forall s \in S \ \bar{s} \in (\overline{S} - \Gamma) \wedge \underline{s} = \varnothing \\
&\Rightarrow \forall q \in \underline{Q_s}, MTTF(I_q) = t_{I_q} + \sum_{k \in out(I_q)} P_{I_q k} \times MTTF(k) \\
&\qquad = 0 + P_{I_q G_q} \times MTTF(G_q) \\
&\qquad = P_{I_q G_q} \times 0 \\
&\qquad = 0 \\
&\Rightarrow MinMTTF(\bar{s}) = \min_{q \in \underline{Q_s}} \{MTTF(I_q)\} = 0. \ \square
\end{aligned}
$$

Therefore, the *attack_surf*$_4$ metric satisfies axiom AS3.

4) Axiom AS4: $\forall s \in S \ \underline{s} \neq \varnothing \Rightarrow [(\forall \bar{t} \in \underline{s} \ MStrength(\bar{t}) = 0) \Rightarrow MStrength(s) = 0]$

$$
\begin{aligned}
&\forall s \in S \ \underline{s} \neq \varnothing, \forall \bar{t} \in \underline{s} \ MinMTTF(\bar{t}) = 0 \\
&\Rightarrow \sum_{\bar{t} \in \underline{s}} MinMTTF(\bar{t}) = 0 \\
&\Rightarrow MinMTTF(s) = \sum_{\bar{t} \in \underline{s}} MinMTTF(\bar{t}) = 0. \ \square
\end{aligned}
$$

Therefore, the *attack_surf*$_4$ metric satisfies axiom AS4.

5) Axiom AS5: $\forall \bar{s} \in \overline{S} \ (\forall t \in \bar{s} \ MStrength(t) = 0) \Rightarrow MStrength(\bar{s}) = 0$

Postulate 41: $\forall \bar{s} \in \overline{S} \ \forall t \in \bar{s} \ MinMTTF(\bar{s}) = \min_{t \in s} \{MinMTTF(t)\}$

$$
\left.
\begin{aligned}
&\forall \bar{s} \in \overline{S} \ \forall t \in \bar{s} \ MinMTTF(t) = 0 \\
&\textit{Postulate } P41
\end{aligned}
\right\}
$$

$$
\begin{aligned}
&\Rightarrow MinMTTF(\bar{s}) = \min_{t \in s} \{MinMTTF(t)\} = 0. \ \square
\end{aligned}
$$

Therefore, the *attack_surf*$_4$ metric satisfies axiom AS5.

6) Axiom AS6: $\forall \bar{s} \in \overline{S}, m \subset \Gamma \ \otimes m \triangleright \bar{s} \Rightarrow MStrength(\bar{s}) \leq \min_{r \in m} \{MStrength(r)\}$

Lemma L41: $\forall \bar{s} \in \overline{S} \ \forall q \in \underline{Q_s}, MTTF(I_q) \ \leq \min_{i \in path(q)} \{MTTF_i(I_q)\}$

Let $i \bullet p$ denote a subpath of path p where $i \bullet p$ ends at the node i of p.

1)$\forall \bar{s} \in \bar{S}, m \subset \Gamma \otimes m \triangleright \bar{s}$

$\Rightarrow (\forall q \in Q_{\bar{s}} \ \forall r_i, r_j \in m \ r_i \neq r_j \Rightarrow r_i, r_j$ *protect* $\bar{s}$ in *different paths of* q)

$\Rightarrow \forall q \in Q_{\bar{s}} \ \forall i \in path(q) \ \forall r \in m,$ if r *is contained in path* i, $MTTF_{r \bullet i}(I_q) = MTTF_i(I_q)$.

2)*Lemma L*41

$$\Rightarrow MinMTTF(\bar{s}) = \underset{q \in Q_{\bar{s}}}{Min}\{MTTF(I_q)\} \leq \underset{q \in Q_{\bar{s}}}{Min}\{\underset{i \in path(q)}{Min}\{MTTF_i(I_q)\}\}$$

Based on 1) ,2), *we can deduct the following* :

$\forall \bar{s} \in \bar{S}, m \subset \Gamma \otimes m \triangleright \bar{s}$

$$\Rightarrow MinMTTF(\bar{s}) \leq \underset{q \in Q_{\bar{s}}}{Min}\{\underset{i \in path(q)}{Min}\{MTTF_{r \bullet i}(I_q)\}\}$$
$$= \underset{r \in m}{Min}\{\underset{q \in Q_{\bar{s}}}{Min}\{MTTF_{r \bullet i}(I_q)\}\}$$
$$= \underset{r \in m}{Min}\{MinMTTF(r)\}. \square$$

Therefore, the *attack_surf*$_4$ metric satisfies axiom AS6.

7) Axiom AS7: $\forall \bar{s} \in \bar{S}, m \subset \Gamma, \oplus m \triangleright \bar{s} \Rightarrow MStrength(\bar{s}) \geq \underset{r \in m}{Max}\{MStrength(r)\}$

*Lemma L*42 : $\forall \bar{s} \in \bar{S} \ \forall q \in Q_{\bar{s}} \ \forall i_1, i_2 \in path(q) \ \ i_1 \subseteq i_2 \Rightarrow MTTF_{i_1}(I_q) \leq MTTF_{i_2}(I_q)$

Let $i \bullet p$ denote a subpath of path p where $i \bullet p$ ends at the node i of p.

(1)$\forall \bar{s} \in \bar{S}, m \subset \Gamma, \oplus m \triangleright \bar{s}$

$\Rightarrow \forall q \in Q_{\bar{s}}, \ \forall$ j $\in$ path(q) $\ \forall r \in m \ r$ *protect* $\bar{s}$ *on path* j

$\Rightarrow \forall q \in Q_{\bar{s}}, \ \forall$ j $\in$ path(q) $\ \forall r \in m \ MTTF_j(I_q) \geq MTTF_{r \bullet j}(I_q)$ (*Lemma L*42)

$\Rightarrow \forall q \in Q_{\bar{s}}, \ \forall$ j $\in$ path(q) $\ MTTF_j(I_q) \geq \underset{r \in m}{Max}\{MTTF_{r \bullet j}(I_q)\}$

(2)$\forall \bar{s} \in \bar{S} \ \forall q \in Q_{\bar{s}} \ \exists!$ j $\in$ path(q) $MTTF(I_q) = MTTF_j(I_q)$

Based on (1) and (2), we can deduct the following:

$\forall \bar{s} \in \bar{S}, m \subset \Gamma, \oplus m \triangleright \bar{s}$

$\Rightarrow \forall q \in Q_{\bar{s}}, \ \exists!$ j $\in$ path(q)

$$MinMTTF(\bar{s}) = \underset{q \in Q_{\bar{s}}}{Min}\{MTTF(I_q)\}$$
$$= \underset{q \in Q_{\bar{s}}}{Min}\{MTTF_j(I_q)\}$$
$$\geq \underset{q \in Q_{\bar{s}}}{Min}\{\underset{r \in m}{Max}\{MTTF_{r \bullet j}(I_q)\}\}$$
$$= \underset{r \in m}{Max}\{\underset{q \in Q_{\bar{s}}}{Min}\{MTTF_{r \bullet j}(I_q)\}\}$$
$$= \underset{r \in m}{Max}\{MinMTTF(r)\}. \square$$

Therefore, the *attack_surf*$_4$ metric satisfies axiom AS7.

In summary, the *attack_surf*$_4$ satisfies axioms AS1~AS7, therefore, can be considered as a mechanism strength metric in our framework.

Appendix B: Theoretical Validation for Sample Metrics based on USIE Model

We present in this section, the detailed proofs for the verification of the sample USIE-based metrics defined in chapter 5.

Metrics *ASD*

$$ASD(U_{cs}) = \frac{\sum_{n_i \in N_{cs}} NumOfServiceDependency(n_i) \times IsAtomicService(n_i)}{\sum_{n_i \in N_{cs}} IsAtomicService(n_i)}$$

Where:

$$IsAtomicService(n_i) = \begin{cases} 1, & if\ n_i\ is\ an\ atomic\ service\ node. \\ 0, & otherwise. \end{cases}$$

$$NumOfServiceDependency(n_i) = The\ number\ of\ service\ nodes\ depending\ directly\ or\ indirectly\ on\ n_i.$$

For *ASD* to be considered as a service complexity metric, it should satisfy axioms AC1~AC5.

1) Axiom AC1: $\forall \bar{s} \in \bar{S}\ Complexity(\bar{s}) \geq 0$

By the definition, *ASD* is non-negative, therefore, satisfies axiom AC1. □

2) Axiom AC2: $\forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ Complexity(\bar{s}) \geq Complexity(t)$

Postulate P 51: $\forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ ASD(\bar{s}) = \underset{t \in \bar{s}}{Max}\{ASD(t)\}$

Postulate P 51 $\Rightarrow$ $\forall \bar{s} \in \bar{S}\ \forall t \in \bar{s}\ ASD(\bar{s}) \geq ASD(t)$ □

Therefore, the *ASD* metric satisfies axiom AC2.

3) Axiom AC3: $\forall s \in S\ \forall \bar{t} \in \underline{s}\ Complexity(\bar{t}) \le Complexity(s)$

Postulate P 52 : $\forall s \in S\ \underline{s} = \varnothing\ \ ASD(s) = ASD(\bar{s}) = 0$

By definition, ASD metric is defined on service configurations whose supporting services are composed only of atomic services, therefore,

Postulate P 52 $\Rightarrow \forall s \in S\ \forall \bar{t} \in \underline{s}\ \ \underline{t} = \varnothing \Rightarrow ASD(\bar{t}) = ASD(t) = 0 \le ASD(s)$ □

Therefore, the *ASD* metric satisfies axiom AC3.

4) Axiom AC4: $\forall s \in S\ \underline{s} = \varnothing \Rightarrow Complexity(\bar{s}) = Complexity(s)$

Postulate P 52 $\Rightarrow \forall s \in S\ \ \underline{s} = \varnothing \Rightarrow ASD(\bar{s}) = ASD(s) = 0$ □

Therefore, the *ASD* metric satisfies axiom AC4.

5) Axiom AC5: $\forall \bar{s}, \bar{t} \in \overline{S}\ \ \bar{s} \subseteq \bar{t} \Rightarrow Complexity(\bar{s}) \le Complexity(\bar{t})$

$$
\begin{aligned}
&\forall \bar{s}, \bar{t} \in \overline{S}\ \ \bar{s} \subseteq \bar{t} \\
&\Rightarrow (\forall h\ h \in \bar{s} \Rightarrow h \in \bar{t}) \\
&\Rightarrow ASD(\bar{s}) = \underset{n \in s}{Max}\{ASD(n)\} \quad (\textit{Postulate P } 51) \\
&\qquad \le \underset{n \in t}{Max}\{ASD(n)\} \\
&\qquad = ASD(\bar{t}) \qquad (\textit{Postulate P } 51) \ \square
\end{aligned}
$$

Therefore, the *ASD* metric satisfies axiom AC5.

In summary, the *ASD* metric satisfies axioms AC1~AC5, therefore, can be considered as a service complexity metric in our framework.

Metrics *Conf*

$$Conf(\bar{s}_1 \to \bar{s}_2) = \frac{1}{1 + count(ILC(\bar{s}_1, \bar{s}_2))}$$

For *Conf* metric to be considered as a service coupling metric, it should satisfy axioms AD1~AD8.

1) Axiom AD1: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \ Coupling(\overline{s_1}, \overline{s_2}) \ge 0$

By definition, $\forall \bar{s}_1, \bar{s}_2 \in \overline{S}\ \ count(ILC(\bar{s}_1, \bar{s}_2)) \ge 0 \Rightarrow \forall \bar{s}_1, \bar{s}_2 \in \overline{S}\ \ Conf(\bar{s}_1 \to \bar{s}_2) \ge 0.$ □ Therefore, the *Conf* metric satisfies axiom AD1.

2) Axiom AD2: $\forall \underline{s_1}, \underline{s_2} \in \underline{S}\ \ \underline{s_1} \propto \underline{s_2} = \varnothing \Rightarrow Coupling(\underline{s_1}, \underline{s_2}) = 0$

$\forall \underline{s_1}, \underline{s_2} \in \underline{S}, \underline{s_1} \propto \underline{s_2} = \varnothing$
$\Rightarrow ILC(\underline{s_1}, \underline{s_2}) = \varnothing$
$\Rightarrow count(ILC(\underline{s_1}, \underline{s_2})) = 0.$
$\Rightarrow Conf(\bar{s}_1 \rightarrow \bar{s}_2) = 1 \square$

Therefore, the *Conf* metric fails Axiom AD2.

3) Axiom AD3: $\forall \underline{s_1}, \underline{s_2} \in \underline{S}\ \ \forall t_1 \in \underline{s_1}\ \ \forall t_2 \in \underline{s_2} \Rightarrow Coupling(\underline{s_1}, \underline{s_2}) \geq Coupling(t_1, t_2)$

$\forall \underline{s_1}, \underline{s_2} \in \underline{S}, \forall t_1 \in \underline{s_1}\ \ \forall t_2 \in \underline{s_2}$
$\Rightarrow ILC(t_1, t_2) \subseteq ILC(\underline{s_1}, \underline{s_2})$
$\Rightarrow count(ILC(t_1, t_2)) \leq count(ILC(\underline{s_1}, \underline{s_2})).$
$\Rightarrow Conf(\bar{s}_1 \rightarrow \bar{s}_2) \leq Conf(\bar{t}_1 \rightarrow \bar{t}_2) \square$

Therefore, the *Conf* metric fails Axiom AD3.

4) Axiom AD4: $\forall t_1, t_2 \in S\ \forall \bar{s}_i \in \underline{t_1}\ \ \forall \bar{s}_j \in \underline{t_2}\ \ Coupling(t_1, t_2) \geq Coupling(\bar{s}_i, \bar{s}_j)$

$\forall t_1, t_2 \in S\ \forall \bar{s}_i \in \underline{t_1}\ \ \forall \bar{s}_j \in \underline{t_2}$
$\Rightarrow ILC(\underline{s_1}, \underline{s_2}) \subseteq ILC(t_1, t_2)$
$\Rightarrow count(ILC(\underline{s_1}, \underline{s_2})) \leq count(ILC(t_1, t_2))$
$\Rightarrow Conf(\bar{s}_1 \rightarrow \bar{s}_2) \geq Conf(t_1 \rightarrow t_2) \square$

Therefore, the *Conf* metric fails Axiom AD4.

5) Axiom AD5: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) \leq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} Coupling(\bar{s}_i, \bar{s}_j)$

$\forall s_1, s_2 \in S\ ILC(s_1, s_2) = \bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} ILC(\bar{s}_i, \bar{s}_j)$
$\Rightarrow count(ILC(s_1, s_2)) = count(\bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} ILC(\bar{s}_i, \bar{s}_j))$
$\Rightarrow count(ILC(s_1, s_2)) \leq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} count(ILC(\bar{s}_i, \bar{s}_j))$
$\Rightarrow Conf(s_1 \rightarrow s_2) \geq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} Conf(\bar{s}_i \rightarrow \bar{s}_j) \square$

Therefore, the *Conf* metric fails Axiom AD5.

6) Axiom AD6: $\forall \underline{s_1}, \underline{s_2}, \bar{s} \in \underline{S}\ \ \underline{s_1} \subseteq \underline{s_2} \Rightarrow Coupling(\underline{s_1}, \bar{s}) \leq Coupling(\underline{s_2}, \bar{s})$

$\forall \overline{s_1}, \overline{s_2}, \overline{s} \in \overline{S}\ \ \overline{s_1} \subseteq \overline{s_2}$
$\Rightarrow ILC(\overline{s_1}, \overline{s}) \subseteq ILC(\overline{s_2}, \overline{s})$
$\Rightarrow count(ILC(\overline{s_1}, \overline{s})) \le count(ILC(\overline{s_2}, \overline{s}))$
$\Rightarrow Conf(\overline{s_1} \to \overline{s}) \ge Conf(\overline{s_2} \to \overline{s})$ □

Therefore, the *Conf* metric fails Axiom AD6.

7) Axiom AD7: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \ Coupling(\overline{s_1}, \overline{s_2}) = Coupling(\overline{s_2}, \overline{s_1})$

By defintion, $\exists \overline{s_1}, \overline{s_2} \in \overline{S}\ \ ILC(\overline{s_1}, \overline{s_2}) \ne ILC(\overline{s_2}, \overline{s_1})$
$\Rightarrow \exists \overline{s_1}, \overline{s_2} \in \overline{S}\ \ count(ILC(\overline{s_1}, \overline{s_2})) \ne count(ILC(\overline{s_2}, \overline{s_1}))$
$\Rightarrow \exists \overline{s_1}, \overline{s_2} \in \overline{S}\ \ Conf(\overline{s_1} \to \overline{s_2}) \ne Conf(\overline{s_2} \to \overline{s_1})$ □

Therefore, the *Conf* metric fails Axiom AD7.

8) Axiom AD8: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) = Coupling(s_2, s_1)$

By defintion, $\exists s_1, s_2 \in S\ \ ILC(s_1, s_2) \ne ILC(s_2, s_1)$
$\Rightarrow \exists s_1, s_2 \in S\ \ \ count(ILC(s_1, s_2)) \ne count(ILC(s_2, s_1))$
$\Rightarrow \exists s_1, s_2 \in S\ \ \ Conf(s_1 \to s_2) \ne Conf(s_2 \to s_1)$ □

Therefore, the *Conf* metric fails Axiom AD8.

In summary, the *Conf* metric cannot be considered as a service coupling metric in our framework since it satisfies only service coupling axioms AD1, but fails to satisfy axioms AD2~AD8.

Metrics *Integ*

$$Integ(\overline{s_1} \to \overline{s_2}) = \frac{1}{1 + count(MC(\overline{s_1}, \overline{s_2}))}$$

For the *Integ* metric to be considered as a service coupling metric, it should satisfy axioms AD1~AD8.

1) Axiom AD1: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \ Coupling(\overline{s_1}, \overline{s_2}) \ge 0$

By definition, $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \ count(MC(\overline{s_1}, \overline{s_2})) \ge 0 \Rightarrow \forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \ Integ(\overline{s_1} \to \overline{s_2}) \ge 0$. □ Therefore, the *Integ* metric satisfies axiom AD1.

2) Axiom AD2: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \ \overline{s_1} \propto \overline{s_2} = \varnothing \Rightarrow Coupling(\overline{s_1}, \overline{s_2}) = 0$

$\forall \overline{s_1}, \overline{s_2} \in \overline{S}, \overline{s_1} \propto \overline{s_2} = \varnothing$

$\Rightarrow MC(\overline{s_1}, \overline{s_2}) = \varnothing$

$\Rightarrow count(MC(\overline{s_1}, \overline{s_2})) = 0.$

$\Rightarrow Integ(\bar{s}_1 \to \bar{s}_2) = 1 \square$

Therefore, the *Integ* metric fails Axiom AD2.

3) Axiom AD3: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \forall t_1 \in \overline{s_1}\ \forall t_2 \in \overline{s_2} \Rightarrow Coupling(\overline{s_1}, \overline{s_2}) \geq Coupling(t_1, t_2)$

$\forall \overline{s_1}, \overline{s_2} \in \overline{S}, \forall t_1 \in \overline{s_1}\ \forall t_2 \in \overline{s_2}$

$\Rightarrow MC(t_1, t_2) \subseteq MC(\overline{s_1}, \overline{s_2})$

$\Rightarrow count(MC(t_1, t_2)) \leq count(MC(\overline{s_1}, \overline{s_2})).$

$\Rightarrow Integ(\bar{s}_1 \to \bar{s}_2) \leq Integ(\bar{t}_1 \to \bar{t}_2) \square$

Therefore, the *Integ* metric fails Axiom AD3.

4) Axiom AD4: $\forall t_1, t_2 \in S\ \forall \bar{s}_i \in \underline{t_1}\ \forall \bar{s}_j \in \underline{t_2}\ Coupling(t_1, t_2) \geq Coupling(\bar{s}_i, \bar{s}_j)$

$\forall t_1, t_2 \in S\ \forall \bar{s}_i \in \underline{t_1}\ \forall \bar{s}_j \in \underline{t_2}$

$\Rightarrow MC(\overline{s_1}, \overline{s_2}) \subseteq MC(t_1, t_2)$

$\Rightarrow count(MC(\overline{s_1}, \overline{s_2})) \leq count(MC(t_1, t_2))$

$\Rightarrow Integ(\bar{s}_1 \to \bar{s}_2) \geq Integ(t_1 \to t_2)\ \square$

Therefore, the *Integ* metric fails Axiom AD4.

5) Axiom AD5: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) \leq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} Coupling(\bar{s}_i, \bar{s}_j)$

$\forall s_1, s_2 \in S\ MC(s_1, s_2) = \bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} MC(\bar{s}_i, \bar{s}_j)$

$\Rightarrow count(MC(s_1, s_2)) = count(\bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} MC(\bar{s}_i, \bar{s}_j))$

$\Rightarrow count(MC(s_1, s_2)) \leq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} count(MC(\bar{s}_i, \bar{s}_j))$

$\Rightarrow Integ(s_1 \to s_2) \geq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} Integ(\bar{s}_i \to \bar{s}_j)\ \square$

Therefore, the *Integ* metric fails Axiom AD5.

6) Axiom AD6: $\forall \overline{s_1}, \overline{s_2}, \bar{s} \in \overline{S}\ \overline{s_1} \subseteq \overline{s_2} \Rightarrow Coupling(\overline{s_1}, \bar{s}) \leq Coupling(\overline{s_2}, \bar{s})$

$\forall \overline{s_1}, \overline{s_2}, \bar{s} \in \overline{S}\ \overline{s_1} \subseteq \overline{s_2}$

$\Rightarrow MC(\overline{s_1}, \bar{s}) \subseteq MC(\overline{s_2}, \bar{s})$

$\Rightarrow count(MC(\overline{s_1}, \bar{s})) \leq count(MC(\overline{s_2}, \bar{s}))$

$\Rightarrow Integ(\bar{s}_1 \to \bar{s}) \geq Integ(\overline{s_2} \to \bar{s})\ \square$

Therefore, the *Integ* metric fails Axiom AD6.

7) Axiom AD7: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ Coupling(\overline{s_1}, \overline{s_2}) = Coupling(\overline{s_2}, \overline{s_1})$

By defintion, $\exists \overline{s_1}, \overline{s_2} \in \overline{S}\ MC(\overline{s_1}, \overline{s_2}) \neq MC(\overline{s_2}, \overline{s_1})$
$\Rightarrow \exists \overline{s_1}, \overline{s_2} \in \overline{S}\ count(MC(\overline{s_1}, \overline{s_2})) \neq count(MC(\overline{s_2}, \overline{s_1}))$
$\Rightarrow \exists \overline{s_1}, \overline{s_2} \in \overline{S}\ Integ(\overline{s_1} \rightarrow \overline{s_2}) \neq Integ(\overline{s_2} \rightarrow \overline{s_1})$ □

Therefore, the *Integ* metric fails Axiom AD7.

8) Axiom AD8: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) = Coupling(s_2, s_1)$

By defintion, $\exists s_1, s_2 \in S\ MC(s_1, s_2) \neq MC(s_2, s_1)$
$\Rightarrow \exists s_1, s_2 \in S\ \ count(MC(s_1, s_2)) \neq count(MC(s_2, s_1))$
$\Rightarrow \exists s_1, s_2 \in S\ \ Integ(s_1 \rightarrow s_2) \neq Integ(s_2 \rightarrow s_1)$ □

Therefore, the *Integ* metric fails Axiom AD8.

In summary, the *Integ* metric cannot be considered as a service coupling metric in our framework since it satisfies only service coupling axioms AD1, but fails to satisfy axioms AD2~AD8.

Metrics *RSR*

$$RSR(U_{s_1}, U_{s_2}) = \frac{cardinality(ResourceNodes(N_{s_1}) \cap ResourceNodes(N_{s_2}))}{cardinality(ResourceNodes(N_{s_1}) \cup ResourceNodes(N_{s_2}))}$$

Where:
$\mathrm{Re}sourceNodes(N_i)$ *returns the set of USIE* Re*source nodes contained in* N_i.

For the *RSR* metric to be considered as a service coupling metric, it should satisfy axioms AD1~AD8.

1) Axiom AD1: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ Coupling(\overline{s_1}, \overline{s_2}) \geq 0$

$\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ RSR(\overline{s_1}, \overline{s_2}) = \frac{cardinality(ResourceNodes(\overline{s_1}) \cap ResourceNodes(\overline{s_2}))}{cardinality(ResourceNodes(\overline{s_1}) \cup ResourceNodes(\overline{s_2}))} \geq 0.$□

Therefore, the *RSR* metric satisfies axiom AD1.

2) Axiom AD2: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \ \overline{s_1} \propto \overline{s_2} = \varnothing \Rightarrow Coupling(\overline{s_1}, \overline{s_2}) = 0$

$\forall \overline{s_1}, \overline{s_2} \in \overline{S}, \overline{s_1} \propto \overline{s_2} = \varnothing$

$\Rightarrow \mathrm{Re}\, sourceNodes(\overline{s_1}) \cap \mathrm{Re}\, sourceNodes(\overline{s_2}) = \varnothing$

$\Rightarrow RSR(\overline{s_1}, \overline{s_2}) = \dfrac{cardinality(\mathrm{Re}\, sourceNodes(\overline{s_1}) \cap \mathrm{Re}\, sourceNodes(\overline{s_2}))}{cardinality(\mathrm{Re}\, sourceNodes(\overline{s_1}) \cup \mathrm{Re}\, sourceNodes(\overline{s_2}))} = 0. \square$

Therefore, the *RSR* metric satisfies axiom AD2.

3) Axiom AD3: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \forall t_1 \in \overline{s_1}\ \forall t_2 \in \overline{s_2} \Rightarrow Coupling(\overline{s_1}, \overline{s_2}) \geq Coupling(t_1, t_2)$

*Postulate P*61: $\forall \overline{s} \in \overline{S}\ \ ResourceNodes(\overline{s}) = \bigcup_{t \in \overline{s}} ResourceNodes(t)$

*Postulate P*61

$\Rightarrow \forall \overline{s_1}, \overline{s_2} \in \overline{S}, \forall t_1 \in \overline{s_1}\ \forall t_2 \in \overline{s_2}$

$\quad ResourceNodes(t_1) \subseteq ResourceNodes(\overline{s_1}) \wedge ResourceNodes(t_2) \subseteq ResourceNodes(\overline{s_2})$

$\Rightarrow ResourceNodes(t_1) \cap ResourceNodes(t_2) \subseteq ResourceNodes(\overline{s_1}) \cap ResourceNodes(\overline{s_2})$

$\Rightarrow cardinality(ResourceNodes(t_1) \cap ResourceNodes(t_2))$

$\quad \leq cardinality(ResourceNodes(\overline{s_1}) \cap ResourceNodes(\overline{s_2}))$

$\Rightarrow RSR(t_1, t_2) \leq RSR(\overline{s_1}, \overline{s_2}). \square$

Therefore, the *RSR* metric satisfies axiom AD3.

4) Axiom AD4: $\forall t_1, t_2 \in S\ \forall \overline{s}_i \in \underline{t_1}\ \forall \overline{s}_j \in \underline{t_2}\ \ Coupling(t_1, t_2) \geq Coupling(\overline{s_i}, \overline{s_j})$

*Postulate P*62: $\forall s \in S\ \ ResourceNodes(s) = \bigcup_{\overline{t} \in \underline{s}} ResourceNodes(\overline{t})$

*Postulate P*62

$\Rightarrow \forall t_1, t_2 \in S\ \forall \overline{s}_i \in \underline{t_1}, \overline{s}_j \in \underline{t_2},$

$\quad ResourceNodes(\overline{s}_i) \subseteq ResourceNodes(t_1) \wedge ResourceNodes(\overline{s}_j) \subseteq ResourceNodes(t_2)$

$\Rightarrow ResourceNodes(\overline{s}_i) \cap ResourceNodes(\overline{s}_j) \subseteq ResourceNodes(t_1) \cap ResourceNodes(t_2)$

$\Rightarrow cardinality(ResourceNodes(\overline{s}_i) \cap ResourceNodes(\overline{s}_j))$

$\quad \leq cardinalityResourceNodes(t_1) \cap ResourceNodes(t_2))$

$\Rightarrow RSR(\overline{s_i}, \overline{s_j}) \leq RSR(t_1, t_2). \square$

Therefore, the *RSR* metric satisfies axiom AD4.

5) Axiom AD5: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) \leq \sum_{\overline{s}_i \in \underline{s_1}} \sum_{\overline{s}_j \in \underline{s_2}} Coupling(\overline{s_i}, \overline{s_j})$

$$\forall s_1, s_2 \in S \;\; RSR(s_1, s_2)$$

$$= \frac{cardinality(\mathrm{Re}\, sourceNodes(s_1) \cap \mathrm{Re}\, sourceNodes(s_2))}{cardinality(\mathrm{Re}\, sourceNodes(s_1) \cup \mathrm{Re}\, sourceNodes(s_2))}$$

$$= \frac{cardinality(\left[\bigcup_{\bar{s}_i \in \underline{s_1}} \mathrm{Re}\, sourceNodes(\bar{s}_i)\right] \cap \left[\bigcup_{\bar{s}_j \in \underline{s_2}} \mathrm{Re}\, sourceNodes(\bar{s}_j)\right])}{cardinality(\left[\bigcup_{\bar{s}_i \in \underline{s_1}} \mathrm{Re}\, sourceNodes(\bar{s}_i)\right] \cup \left[\bigcup_{\bar{s}_j \in \underline{s_2}} \mathrm{Re}\, sourceNodes(\bar{s}_j)\right])} \quad (\textit{Postulate } P62)$$

$$= \frac{cardinality(\left(\bigcup_{\bar{s}_i \in \underline{s_1}, \bar{s}_j \in \underline{s_2}} \left[\mathrm{Re}\, sourceNodes(\bar{s}_i) \cap \mathrm{Re}\, sourceNodes(\bar{s}_j)\right]\right))}{cardinality(\left[\bigcup_{\bar{s}_i \in \underline{s_1}} \mathrm{Re}\, sourceNodes(\bar{s}_i)\right] \cup \left[\bigcup_{\bar{s}_j \in \underline{s_2}} \mathrm{Re}\, sourceNodes(\bar{s}_j)\right])}$$

$$\leq \frac{\sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} cardinality(\mathrm{Re}\, sourceNodes(\bar{s}_i) \cap \mathrm{Re}\, sourceNodes(\bar{s}_j))}{cardinality(\left[\bigcup_{\bar{s}_i \in \underline{s_1}} \mathrm{Re}\, sourceNodes(\bar{s}_i)\right] \cup \left[\bigcup_{\bar{s}_j \in \underline{s_2}} \mathrm{Re}\, sourceNodes(\bar{s}_j)\right])}$$

$$\leq \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} \frac{cardinality(\mathrm{Re}\, sourceNodes(\bar{s}_i) \cap \mathrm{Re}\, sourceNodes(\bar{s}_j))}{cardinality(\mathrm{Re}\, sourceNodes(\bar{s}_i) \cup \mathrm{Re}\, sourceNodes(\bar{s}_j))}$$

$$= \sum_{\bar{s}_i \in \underline{s_1}} \sum_{\bar{s}_j \in \underline{s_2}} RSR(\bar{s}_i, \bar{s}_j) \;\square$$

Therefore, the *RSR* metric satisfies axiom AD5.

6) Axiom AD6: $\forall \bar{s_1}, \bar{s_2}, \bar{s} \in \bar{S} \;\; \bar{s_1} \subseteq \bar{s_2} \Rightarrow Coupling(\bar{s_1}, \bar{s}) \leq Coupling(\bar{s_2}, \bar{s})$

$$\forall \bar{s_1}, \bar{s_2}, \bar{s} \in \bar{S} \;\; \bar{s_1} \subseteq \bar{s_2}$$

$$\Rightarrow \mathrm{Re}\, sourceNodes(\bar{s_1}) \subseteq \mathrm{Re}\, sourceNodes(\bar{s_2})$$

$$\Rightarrow \mathrm{Re}\, sourceNodes(\bar{s_1}) \cap \mathrm{Re}\, sourceNodes(\bar{s})$$

$$\subseteq \mathrm{Re}\, sourceNodes(\bar{s_2}) \cap \mathrm{Re}\, sourceNodes(\bar{s})$$

$$\Rightarrow cardinality(\mathrm{Re}\, sourceNodes(\bar{s_1}) \cap \mathrm{Re}\, sourceNodes(\bar{s}))$$

$$\leq cardinality(\mathrm{Re}\, sourceNodes(\bar{s_2}) \cap \mathrm{Re}\, sourceNodes(\bar{s}))$$

$$\Rightarrow RSR(\bar{s_1}, \bar{s}) \leq RSR(\bar{s_2}, \bar{s}). \;\square$$

Therefore, the *RSR* metric satisfies axiom AD6.

7) Axiom AD7: $\forall \bar{s_1}, \bar{s_2} \in \bar{S} \;\; Coupling(\bar{s_1}, \bar{s_2}) = Coupling(\bar{s_2}, \bar{s_1})$

$$\forall \bar{s_1}, \bar{s_2} \in \bar{S}$$

$$\mathrm{Re}\, sourceNodes(\bar{s_1}) \cap \mathrm{Re}\, sourceNodes(\bar{s_2}) = \mathrm{Re}\, sourceNodes(\bar{s_2}) \cap \mathrm{Re}\, sourceNodes(\bar{s_1}),$$

$$\mathrm{Re}\, sourceNodes(\bar{s_1}) \cup \mathrm{Re}\, sourceNodes(\bar{s_2}) = \mathrm{Re}\, sourceNodes(\bar{s_2}) \cup \mathrm{Re}\, sourceNodes(\bar{s_1})$$

$$\Rightarrow RSR(\bar{s_1}, \bar{s_2}) = RSR(\bar{s_2}, \bar{s_1}). \;\square$$

Therefore, the *RSR* metric satisfies axiom AD7.

8) Axiom AD8: $\forall s_1, s_2 \in S\ Coupling(s_1, s_2) = Coupling(s_2, s_1)$

$\forall s_1, s_2 \in S$

$\mathrm{Re}\, sourceNodes(s_1) \cap \mathrm{Re}\, sourceNodes(s_2) = \mathrm{Re}\, sourceNodes(s_2) \cap \mathrm{Re}\, sourceNodes(s_1),$

$\mathrm{Re}\, sourceNodes(s_1) \cup \mathrm{Re}\, sourceNodes(s_2) = \mathrm{Re}\, sourceNodes(s_2) \cup \mathrm{Re}\, sourceNodes(s_1)$

$\Rightarrow RSR(s_1, s_2) = RSR(s_2, s_1).$ □

Therefore, the *RSR* metric satisfies axiom AD8.

In summary, the *RSR* satisfies axioms AD1~AD8, therefore, can be considered as a service coupling metric in our framework.

Metrics *EPC*

$$EPC(\bar{s}) = count(Actual\ \mathrm{Pr}\, ivileges(\bar{s}) - Least\ \mathrm{Pr}\, ivileges(\bar{s}))$$

For the metric *EPC* to be considered as an excess privilege metric, it should satisfy axioms AP1~AP5.

1) Axiom AP1: $\forall \bar{s} \in \overline{S}\ EP(\bar{s}) \geq 0$

$\forall \bar{s} \in \overline{S},\ EPC(\bar{s}) = count(Actual\ \mathrm{Pr}\, ivileges(\bar{s}) - Least\ \mathrm{Pr}\, ivileges(\bar{s})) \geq 0.$ □

Therefore, the *EPC* metric satisfies axiom AP1.

2) Axiom AP2: $\forall s \in S\ \forall \bar{t} \in \underline{s}\ EP(s) \geq EP(\bar{t})$

*Postulate P*71:

$$\forall s \in S\ \ Actual\ \mathrm{Pr}\, ivileges(s) = \bigcup_{\bar{t} \in s} Actual\ \mathrm{Pr}\, ivileges(\bar{t}), Least\ \mathrm{Pr}\, ivileges(s) = \bigcup_{\bar{t} \in s} Least\ \mathrm{Pr}\, ivileges(\bar{t})$$

$$\forall s \in S\ \forall \bar{t} \in \underline{s}\ \ EPC(s) = count(Actual\ \mathrm{Pr}\, ivileges(s) - Least\ \mathrm{Pr}\, ivileges(s))$$

$$= count(\bigcup_{\bar{t} \in s} Actual\ \mathrm{Pr}\, ivileges(\bar{t}) - \bigcup_{\bar{t} \in s} Least\ \mathrm{Pr}\, ivileges(\bar{t})) \quad (postulate\ P71)$$

$$= count(\bigcup_{\bar{t} \in s} (Actual\ \mathrm{Pr}\, ivileges(\bar{t}) - Least\ \mathrm{Pr}\, ivileges(\bar{t})))$$

$$\geq count(Actual\ \mathrm{Pr}\, ivileges(\bar{t}) - Least\ \mathrm{Pr}\, ivileges(\bar{t}))$$

$$= EPC(\bar{t})$$ □

Therefore, the *EPC* metric satisfies axiom AP2.

3) Axiom AP3: $\forall s \in S\ \underline{s} \neq \varnothing \Rightarrow EP(s) \leq \sum_{\bar{t}_i \in \underline{s}} EP(\bar{t}_i)$

$$\forall s \in S \ \underline{s} \neq \varnothing \ \ EPC(s) = count(Actual\ \Pr ivileges(s) - Least\ \Pr ivileges(s))$$
$$= count(\bigcup_{\bar{t} \in s} Actual\ \Pr ivileges(\bar{t}) - \bigcup_{\bar{t} \in s} Least\ \Pr ivileges(\bar{t})) \quad (postulate\ P71)$$
$$= count(\bigcup_{\bar{t} \in s} (Actual\ \Pr ivileges(\bar{t}) - Least\ \Pr ivileges(\bar{t})))$$
$$\leq \sum_{\bar{t} \in s} count(Actual\ \Pr ivileges(\bar{t}) - Least\ \Pr ivileges(\bar{t}))$$
$$= \sum_{\bar{t} \in s} EPC(\bar{t}) \square$$

Therefore, the *EPC* metric satisfies axiom AP3.

4) Axiom AP4: $\forall s \in S \ EP(\bar{s}) \geq EP(s)$

*Postulate P*72 :

$$\forall \bar{s} \in \bar{S} \ \ Actual\ \Pr ivileges(\bar{s}) = \bigcup_{t \in \bar{s}} Actual\ \Pr ivileges(t), Least\ \Pr ivileges(\bar{s}) = \bigcup_{t \in \bar{s}} Least\ \Pr ivileges(t)$$

$$\forall s \in S \ \ EPC(\bar{s}) = count(Actual\ \Pr ivileges(\bar{s}) - Least\ \Pr ivileges(\bar{s}))$$
$$= count(\bigcup_{s \in \bar{s}} Actual\ \Pr ivileges(s) - \bigcup_{s \in \bar{s}} Least\ \Pr ivileges(s)) \quad (postulate\ P72)$$
$$= counts\bigcup_{s \in \bar{s}} (Actual\ \Pr ivileges(s) - Least\ \Pr ivileges(s)))$$
$$\geq count(Actual\ \Pr ivileges(s) - Least\ \Pr ivileges(s))$$
$$= EPC(s) \square$$

Therefore, the *EPC* metric satisfies axiom AP4.

5) Axiom AP5: $\forall s \in S \ \underline{s} = \varnothing \Rightarrow EP(\bar{s}) = EP(s)$

$$\forall s \in S \ \underline{s} \ = \varnothing$$
$$\Rightarrow s = \bar{s}$$
$$\Rightarrow (Actual\ \Pr ivileges(s) - Least\ \Pr ivileges(s)) = \ (Actual\ \Pr ivileges(\bar{s}) - Least\ \Pr ivileges(\bar{s}))$$
$$\Rightarrow EPC(s) = EPC(\bar{s}) \ \square$$

Therefore, the *EPC* metric satisfies axiom AP5.

In summary, the *EPC* metric satisfies axioms AP1~AP5, therefore, can be considered as an excess privilege metric in our framework.

Metrics *PMS*

$$PMS(p_r, \bar{s}) = \underset{i \in PMP(\bar{s}) \ \& \ p_r \in \Pr ivileges(i)}{Min} \{MechanismStrength(i)\}$$

For the metric *PMS* to be considered as a mechanism strength metric, it should satisfy axioms AS1~AS7.

1) Axiom AS1: $\forall \bar{s} \in \bar{S} \;\; MStrength(\bar{s}) \geq 0$

$\forall \bar{s} \in \bar{S} \;\; \forall i \in PMP(\bar{s}) \;\; MechanismStrength(i) \geq 0$

$\Rightarrow PMS(p_r, \bar{s}) = \underset{i \in PMP(\bar{s}) \;\&\; p_r \in \Pr ivileges(i)}{Min} \{MechanismStrength(i)\} \geq 0.$ □

Therefore, the *PMS* metric satisfies axiom AS1.

2) Axiom AS2: $\forall s \in S \;\; \underline{s} = \varnothing \Rightarrow MStrength(\bar{s}) = MStrength(s)$

$\forall s \in S \;\; \underline{s} = \varnothing \Rightarrow s = \bar{s} \Rightarrow PMS(p_r, s) = PMS(p_r, \bar{s})$ □

Therefore, the *PMS* metric satisfies axiom AS2.

3) Axiom AS3: $\forall s \in S \;\; [\bar{s} \in (\bar{S} - \Gamma) \wedge \underline{s} = \varnothing] \;\; \Rightarrow MStrength(\bar{s}) = 0$

$\forall s \in S \;\; [\bar{s} \in (\bar{S} - \Gamma) \wedge \underline{s} = \varnothing]$

$\Rightarrow \forall i \in PMP(\bar{s}), \; MechanismStrength(i) = 0$

$\Rightarrow PMS(p_r, \bar{s}) = \underset{i \in PMP(\bar{s}) \;\&\; p_r \in \Pr ivileges(i)}{Min} \{MechanismStrength(i)\} = 0$ □

Therefore, the *PMS* metric satisfies axiom AS3.

4) Axiom AS4: $\forall s \in S \;\; \underline{s} \neq \varnothing \Rightarrow [(\forall \bar{t} \in \underline{s} \;\; MStrength(\bar{t}) = 0) \Rightarrow MStrength(s) = 0]$

$\forall s \in S \;\; \underline{s} \neq \varnothing, \forall \bar{t} \in s \;\; PMS(p_r, \bar{t}) = 0$

$\Rightarrow PMS(p_r, s) \leq Max\{PMS(p_r, s)\}$

$= \sum_{t \in s} PMS(p_r, \bar{t})$

$= 0$ □

Therefore, the *PMS* metric satisfies axiom AS4.

5) Axiom AS5: $\forall \bar{s} \in \bar{S} \;\; \forall t \in \bar{s} \;\; MStrength(t) = 0 \Rightarrow MStrength(\bar{s}) = 0$

*Postulate P*81: $\forall \bar{s} \in \bar{S} \;\; PMS(p_r, \bar{s}) = \underset{t \in s}{Min}\{PMS(p_r, t)\}$

$\left.\begin{array}{l} \forall \bar{s} \in \bar{S} \;\; \forall t \in \bar{s} \;\; PMS(p_r, t) = 0 \\ Postulate\ P81 \end{array}\right\}$

$\Rightarrow PMS(p_r, \bar{s}) = \underset{t \in s}{Min}\{PMS(p_r, t)\} = 0.$ □

Therefore, the *PMS* metric satisfies axiom AS5.

6) Axiom AS6: $\forall \bar{s} \in \bar{S}, m \subset \Gamma, \otimes m \rhd \bar{s} \Rightarrow MStrength(\bar{s}) \leq \underset{r \in m}{Min}\{MStrength(r)\}$

*Postulate P*82 : $\forall m \subset \Gamma \ \forall r \in m \ PMS(p_r, r) = MechanismStrength(r)$

$\forall \bar{s} \in \bar{S}, m \subset \Gamma, \otimes m \rhd \bar{s}$

$\Rightarrow \otimes m = Mechanism(PMP(\bar{s}))$

$$\Rightarrow PMS(p_r, \bar{s}) = \min_{i \in PMP(\bar{s}) \ \& \ p_r \in Privileges(i)} \{MechanismStrength(i)\}$$

$$= \min_{r \in Mechanism(PMP(s)) \ \& \ p_r \in Privileges(i)} \{MechanismStrength(r)\}$$

$$= \min_{r \in \otimes m \ \& \ p_r \in Privileges(i)} \{MechanismStrength(r)\}$$

$$\leq \min_{r \in m} \{MechanismStrength(r)\}$$

$$= \min_{r \in m} \{PMS(p_r, r)\} \qquad (Postulate\ P81) \ \square$$

Therefore, the *PMS* metric satisfies axiom AS6.

7) Axiom AS7: $\forall \bar{s} \in \bar{S}, m \subset \Gamma, \oplus m \rhd \bar{s} \Rightarrow MStrength(\bar{s}) \geq \max_{r \in m} \{MStrength(r)\}$

$\forall \bar{s} \in \bar{S}, m \subset \Gamma, \oplus m \rhd \bar{s}$

$\Rightarrow \oplus m = Mechanism(PMP(\bar{s}))$

$$\Rightarrow PMS(p_r, \bar{s}) = \min_{i \in PMP(\bar{s}) \ \& \ p_r \in Privileges(i)} \{MechanismStrength(i)\}$$

$$= \min_{r \in Mechanism(PMP(s)) \ \& \ p_r \in Privileges(i)} \{MechanismStrength(r)\}$$

$$= \min_{r \in \oplus m \ \& \ p_r \in Privileges(i)} \{MechanismStrength(r)\}$$

$$\geq \max_{r \in m} \{MechanismStrength(r)\}$$

$$= \max_{r \in m} \{PMS(p_r, r)\} \qquad (Postulate\ P81) \ \square$$

Therefore, the *PMS* metric satisfies axiom AS7.

In summary, the *PMS* satisfies axioms AS1~AS7, therefore, can be considered as a mechanism strength metric in our framework.

Appendix C: Framework Automation: the STEM Toolkit

We present in this section a software toolkit that implements our software attackability analysis framework. The tool, named STEM for *Security Testing and Engineering using Metrics* is currently under development. We first present an overview of the tool, and then introduce briefly its major components and functionalities.

Overview

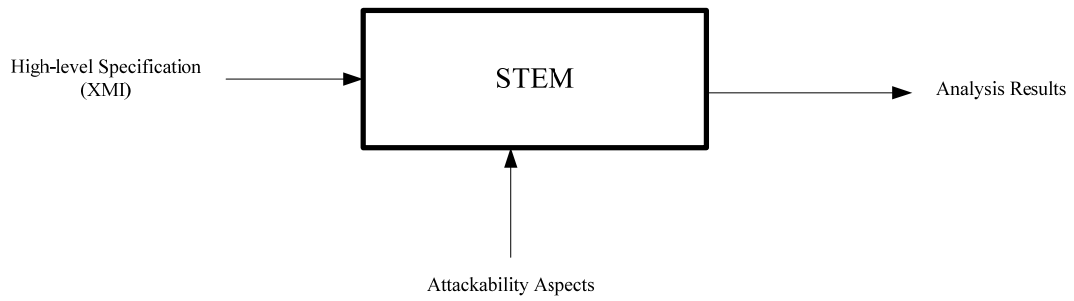

Figure C.1 High Level Architecture of STEM

The main objective of STEM is to allow software architects to evaluate various attackability attributes of the design of a software application based on automatically generated security metrics. STEM receives as input a UML specification of a software application using XML Metadata Interchange (XMI) format. XMI is a XML notation that uses a model exchange format supported by most UML modeling tools (e.g. Rational Rose, ArgoUML, Poseidon, etc.). STEM processes the behavioral information of a software application and derives a set of security metrics. Finally, STEM presents the attackability results based on analysis of the security metrics computed. Figure C.1 illustrates the high level structure of the STEM tool.

The current version of STEM provides some nice features. For instance, it provides a graphical user interface; it allows a user to select various attackability attributes for analysis; it provides a facility to import UML diagrams via XMI format. The development of the STEM toolkit is ongoing and will progressively include the results obtained in our research.

STEM Main Interface

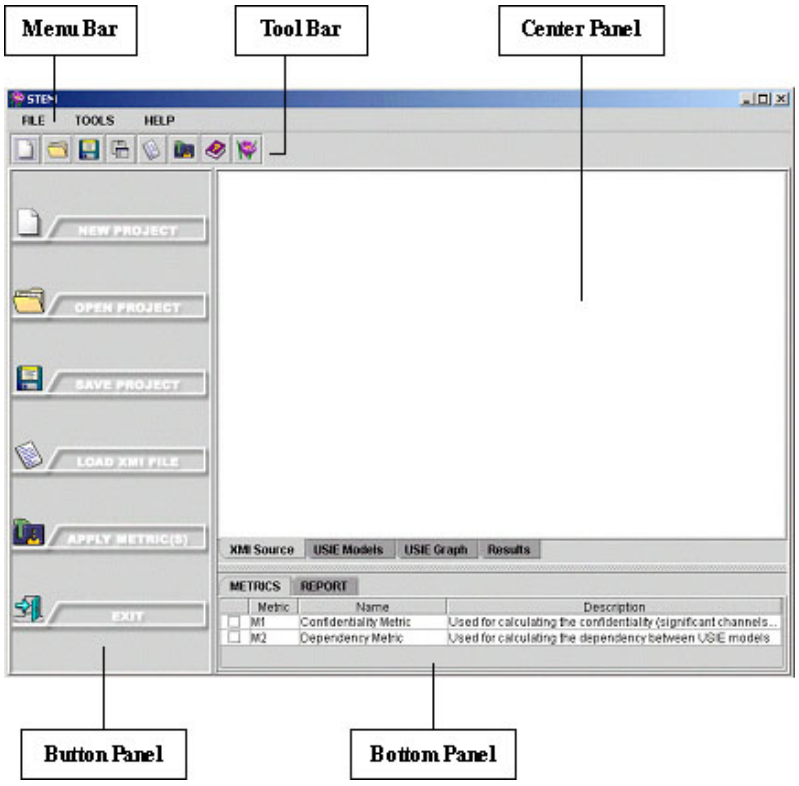

Figure C.2. The Main Interface of STEM

A snapshot of current STEM main interface is shown in Figure C.2. The main interface consists of several components including menu bar, tool bar, centre panel, button panel and bottom panel. The functionalities implemented for these components are described as follows:

Menu Bar: The menu bar consists of three main menus, namely file menu, tools menu and help menu. The file menu contains several menu items each of which corresponds to specific file management function such as creating a new project, saving a project or exiting the project etc. The tools menu is composed of menu items for loading and parsing an XMI file. The help menu simply contains a menu item linking to the user manual.

Tool Bar: The tool bar simply consists of buttons that will trigger the same functions as the menu items contained in the menu bar. Specifically, there are eight buttons placed in the toolbar (in order): 1) Button for creating a new project. 2) Button for opening an existing project. 3) Button for saving the current project. 4) Button for saving the current project to a file. 5) Button for loading an XMI file. 6) Button for applying the selected metrics. 7) Button for opening the user manual in the browser. 8) Button for displaying information about STEM.

Button Panel: The button panel provides several quick access button components that implement the same functionalities as the buttons in the tool bar.

Centre Panel: The centre panel is a display panel that consists of four sub panes including *XMI Source* pane, *USIE Models* pane, *USIE Graphs* pane, and *Results* pane. Specifically, the *XMI Source* panel shows the contents of the XMI file loaded into the STEM tool. The *USIE Models* pane is used to display the USIE semantic models contained in the XMI file. The *USIE Graphs* pane is used simply to display the corresponding USIE graph generated for the USIE models of the project. The *Results* pane will show the attackability reports calculated from the metrics predefined in the STEM tool.

Bottom Panel: The bottom panel includes a metrics pane and a report pane. The metrics pane displays all the attackability attributes that can be analyzed in the STEM tool. The analysis metrics for the attackability attributes are predefined and embedded information in the STEM tool. The report pane is a neat area on the STEM main interface, which

displays all the commands executed by the user with all the sub steps that happen within a command. Using the report pane, STEM user is able to track the execution status of the tool and receive some runtime information. For instance, a snapshot of the *Report* pane showing runtime information is shown in Figure C.3.

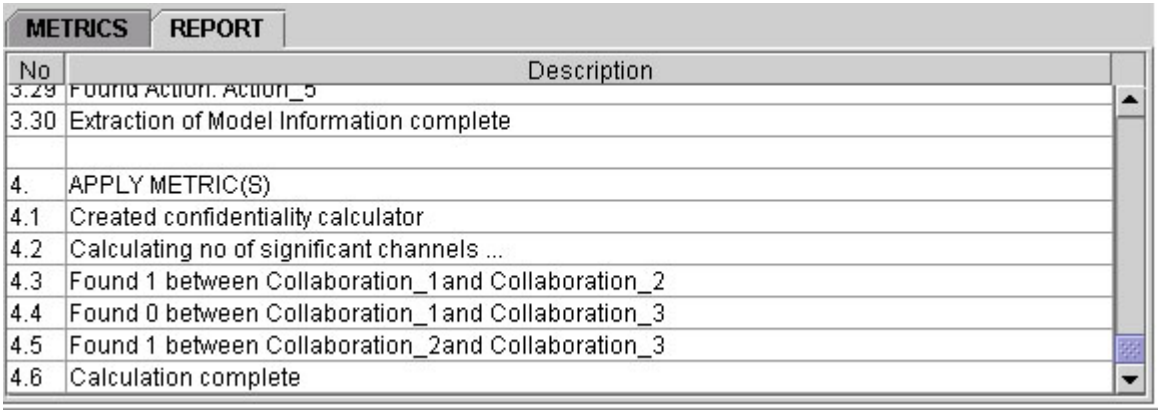

Figure C.3. Runtime Status Display

STEM Functionality

The primary objective of STEM is to generate automatically USIE model from high level system specifications and apply predefined metrics to generate software attackability reports. In this section, we explain how to use STEM to conduct these two activities.

Generate USIE Model: STEM doesn't provide an environment for users to specify directly software high level designs, but require a design specification file in XMI format as the input. Many prevalent software modeling and designing tools, such as argoUML and Poseidon, can export the graphical information into a XMI format. The current STEM users need to use these existing software modeling tools to create high level design artifacts required for STEM analysis and import the XMI file into STEM. Specifically, STEM requires the user-system interaction information to be modeled in UML sequence diagram and contained in the XMI file. Once the user-system interaction design is

completed and exported into a XMI format, STEM users can follow three steps as follows to generate USIE models for the software design.

1) Open a project: A STEM project needs to be created firstly before using STEM functions. On the STEM main interface, STEM users can simply create a new project by clicking on "*File->new project*" from the menu bar or open an existing project by clicking on the "*Open Project*" button from the button panel. The project information can be specified in the pop up window appeared. For instance, Figure C.4 shows the process of STEM tool to open an existing project

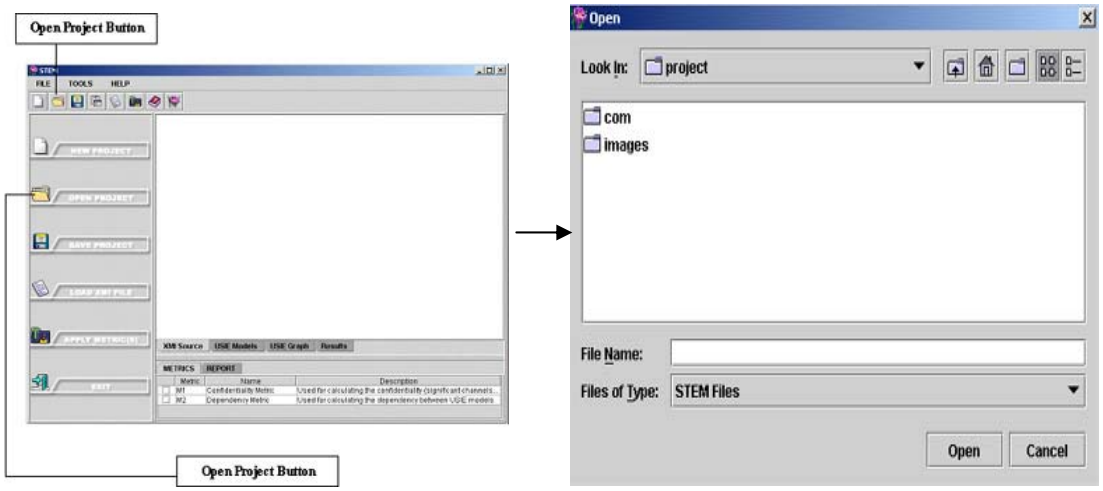

Figure C.4. Open a STEM project

2) Load XMI file: loading a XMI file in STEM can be achieved simply by clicking on the "*Load XMI File*" button on the button panel or select the same option from the toolbar or even the menu bar. Once the button is pressed, a pop-up window appears asking the user to select the XMI file. After the file has been selected and the *"OK"* button is pressed, the XMI file is loaded into STEM. Note that loading an XMI file will clear the contents of a previously loaded file (including its graph, models and results). Figure C.5 demonstrates the process of loading a XMI file into STEM.

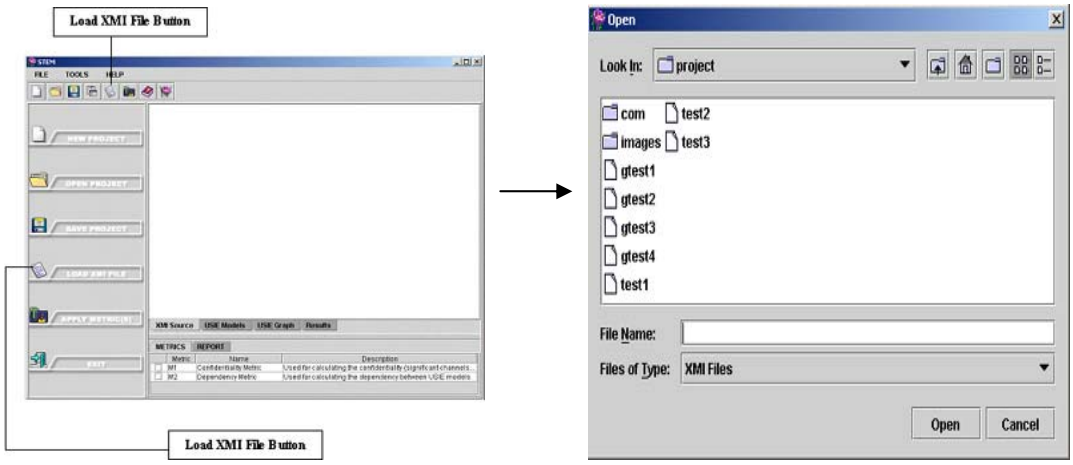

Figure C.5. Load a XMI file

3) Display USIE Models: Once the XMI file is loaded, STEM will automatically parse the XMI file and derive USIE models from the user-system interaction information embedded in the XMI file.

STEM users can view the original XMI file from the *XML Source* pane of the centre display penal. Figure C.6 shows a snapshot of the *XMI Source* pane displaying XMI information.

The USIE models generated can be viewed from the *USIE Models* pane of the centre panel. A hierarchical structure is presented when STEM displays the USIE models. The root node of the hierarchical structure represents the project itself. Each USIE model node can be further expanded to show USIE nodes and edges involved. For instance, Figure C.7 shows an example USIE model displayed in the STEM *USIE Models* pane.

The USIE model in graphical forms can be viewed from the *USIE Graphs* pane of the centre panel. The graph can be zoomed in and out using the two corresponding buttons shown on the upper left part of the *USIE Graphs* pane.

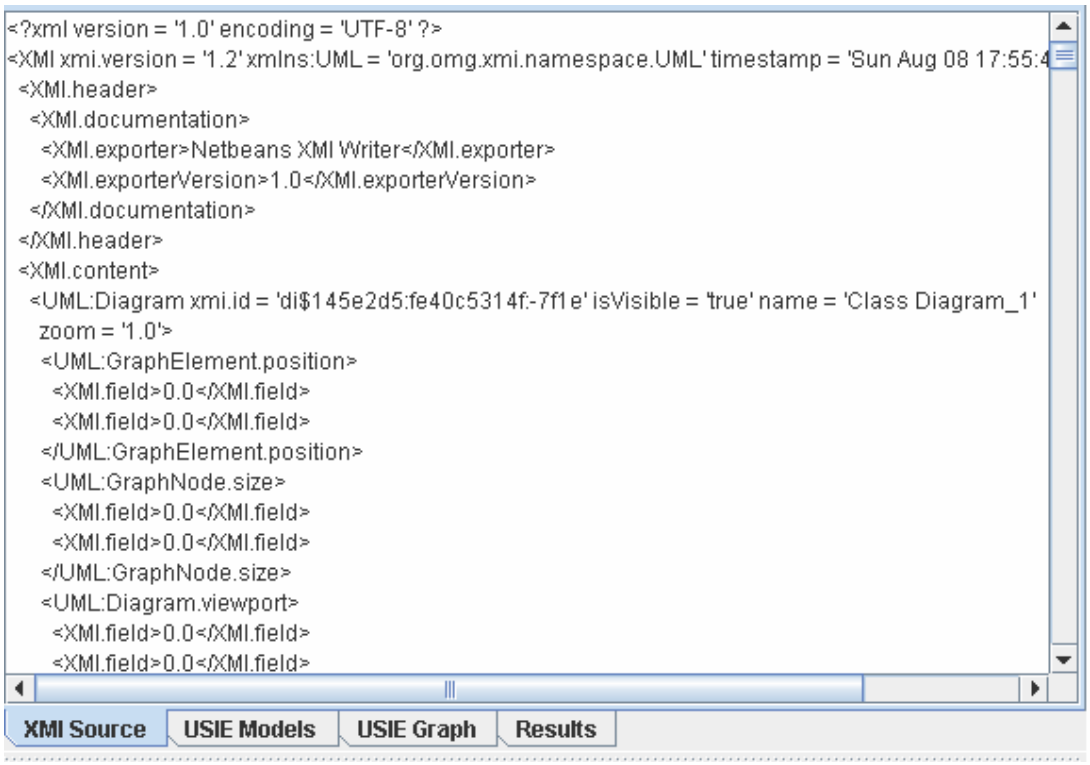

Figure C.6. XMI Source Display

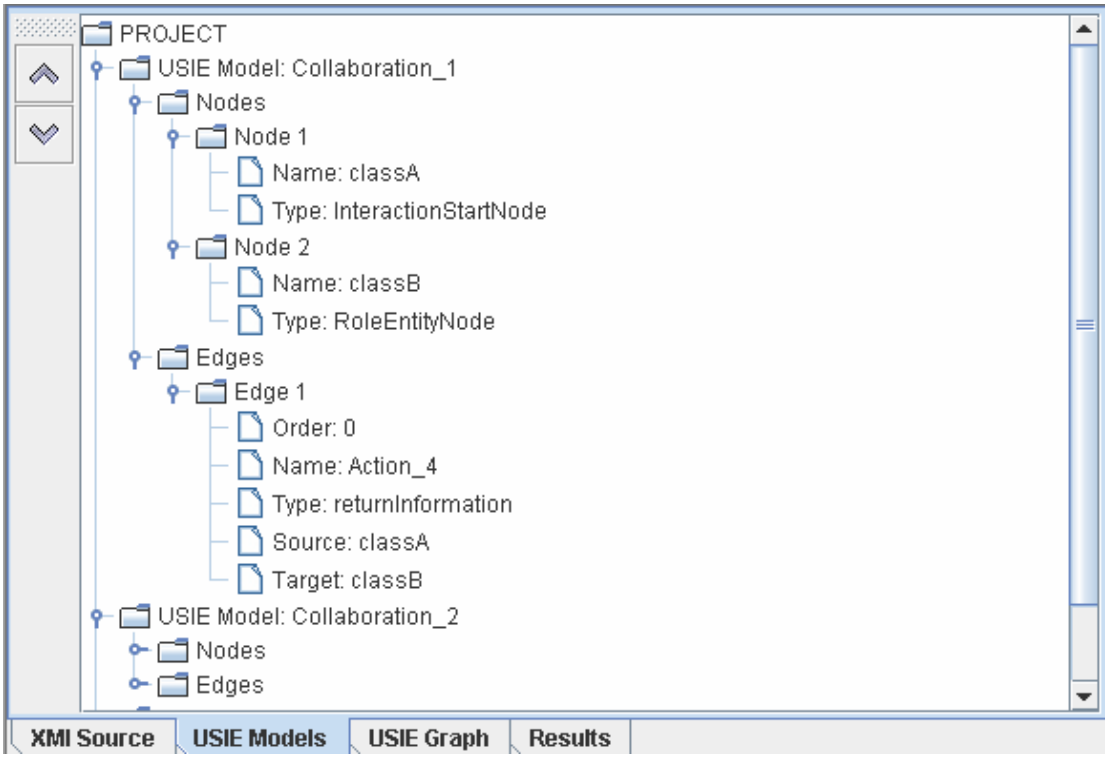

Figure C.7. USIE Model Information

Generate Attackability Report: STEM allows users to specify attackability perspectives to generate reports based on corresponding metrics. Three steps are involved in the reporting process. Specifically, STEM users need to firstly specify the attackability perspectives they are interested in from the STEM attackability library; secondly invoke the STEM tool to calculate corresponding metrics and generate reports; and thirdly go to the results pane to view the reports. We illustrate the three steps in details as follows.

1) Specify attackability perspectives: the *Metrics* pane of STEM tool displays all the attackability aspects that can be analyzed in the current version of STEM attackability library. STEM users simply need to checkmark the attackability perspectives that he/she wants to analyze for the software design. The attackability report will be generated according to STEM user's attackability selections. A snapshot of the *Metrics* pane is shown in Figure C.8.

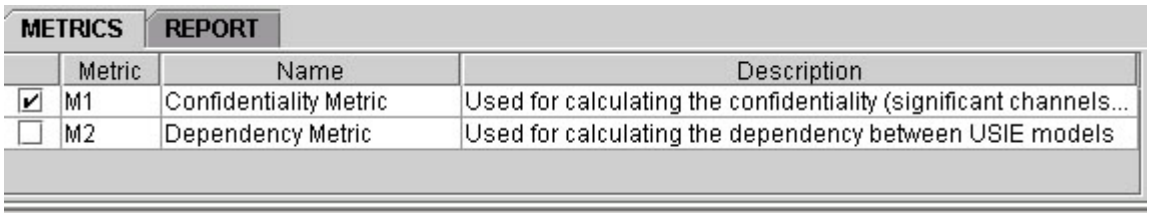

Figure C.8. Attackability Selection

2) Generate Report: Once the STEM user has selected the attackability attributes, he can start generating corresponding attackability report by either clicking on the *Apply Metrics Button* on the tool bar or clicking on the *Apply Metrics* button on the button panel. Figure C.9 shows the button locations on STEM main interface.

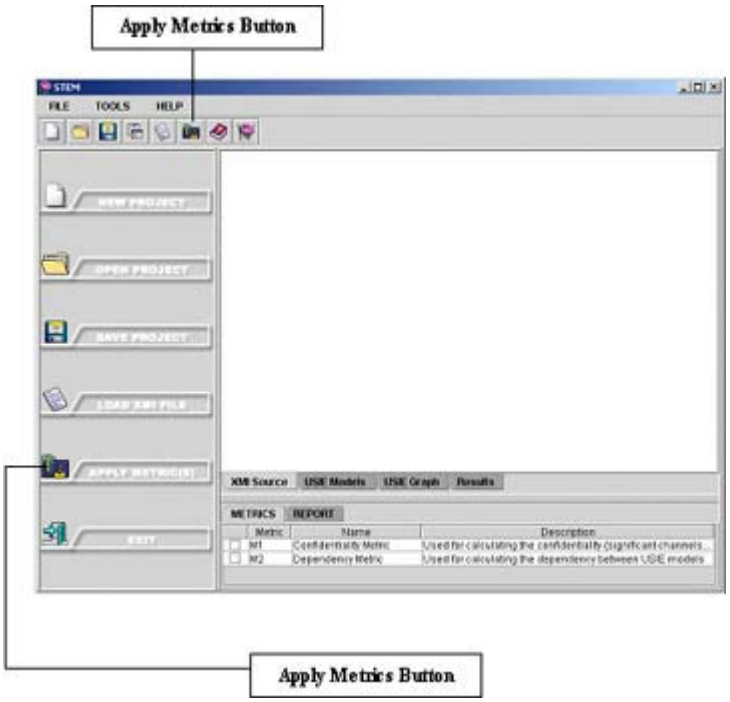

Figure C.9.Report Generation

3) Display Report: The attackability report will be shown in the *Result* pane of the centre display panel. STEM users need to click on the *Result* tab located under the bottom of the centre panel. The report is generally displayed in a table format. The content of the report varies according to the selected attackability attributes. For example, Figure C.10 illustrates a sample report for confidentiality analysis.

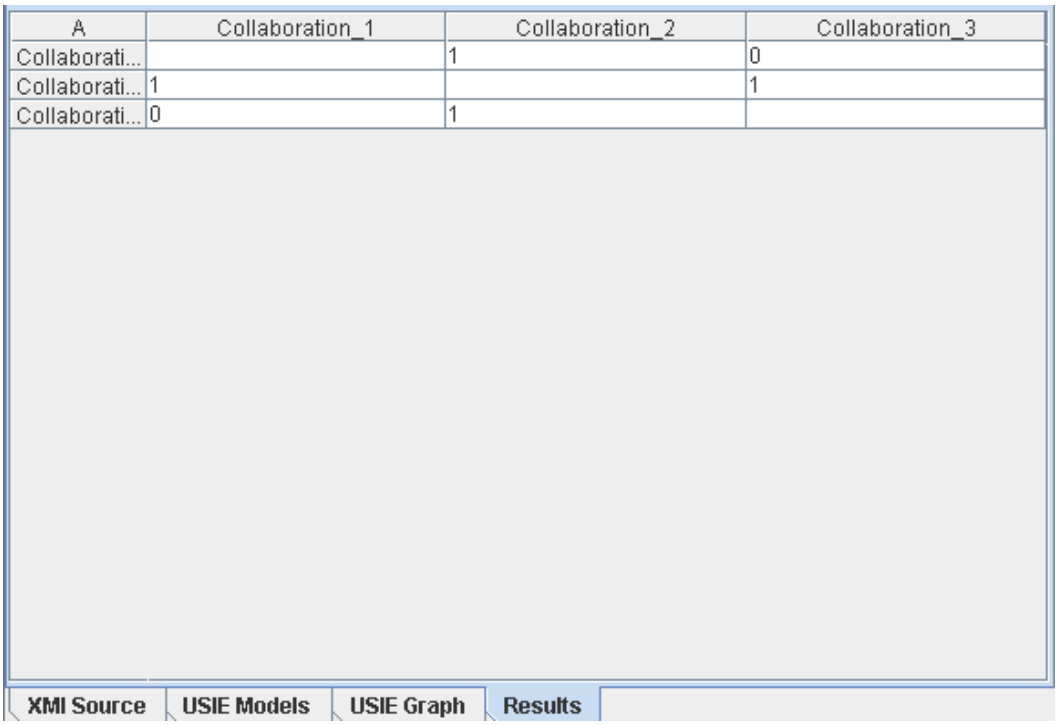

Figure C.10. Report Display

Summary

In this section, we have presented the STEM toolkit. STEM is an ongoing project currently being carried in the Information Security and Object Technology (ISOT) group. The STEM toolkit can assist USIE-based security analysis in generating automatically USIE models from UML diagrams and computing and analyzing various security metrics. The current version of the tool implements the algorithms introduced in this dissertation for model building and analysis. The development of the tool will continue by integrating new analysis approaches and metrics as they are defined.