Applied Mathematics & Information Sciences **1**(2) (2007), 129-156

Properties for Security Measures of Software Products

Yanguo Michael Liu and Issa Traoré

Received February 10, 2007

University of Victoria, P.O.Box 3055 STN CSC,

Victoria, B.C., V8W 3P6, Canada

E-mail: {yliu, itraore}@ece.uvic.ca

A large number of attacks on computing systems succeed because of the existence of software flaws (e.g. buffer overflow, race conditions etc.) that could be fixed through a careful design process. An effective way of improving the quality of software products consists of using metrics to guide the development process. The field of software security metrics however is still in infancy in contrast with the area of traditional software metrics such as reliability metrics for which several key results have been obtained so far. We identify in this paper a number of internal software attributes that could be related to a variety of security qualities. Since theoretical validation is an important step in the development of any metrics program, we focus in this paper on studying the measurement properties associated with these internal attributes. The properties, based on popular security design principles in use in security engineering processes, can be used to guide the search of software security metrics. We study the feasibility of our theoretical framework by presenting case studies based on metrics derived from existing security measurement frameworks, namely the attack surface metrics system and the privilege graph paradigm.

Keywords: Software security, software metrics, security metrics, quality engineering, quality attributes, requirements engineering.

1 Rationale and Problem Statement

Metrics are needed to evaluate current level of security, to identify resources for improvements and to implement those improvements [7]. Vaughn, however, questions the feasibility of "measures and metrics for trusted information systems" [21]. 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 software engineering discipline is not compliant with the laws of physics and

faces huge challenges in establishing correctness. Vaughn, however, suggests that effective security metrics can be defined by accepting some risk in how they are used and by validating them in the real world through empirical investigation and experimentation. Likewise most previous works on software metrics adopt an empirical approach for their definition and their evaluation. Also, as suggested by Vaughn, the need for identifying similar laws like in physics for theoretical validation of software metrics has been widely recognized [4], [5], [8], [11], [21], [23].

Fenton identifies three classes of entities that are of interest of software engineering measurement, namely products, processes and resources [4]. We are interested in this work on developing and validating security metrics at the software product level. Security is a multifaceted quality concept, in which each facet represents a separate external software attribute in its own [15]. In general, external attributes are not directly related to any feature of the software product. However, to improve the software product we need to be able to affect its internal features. So we need to identify some internal attributes, which influence directly or indirectly software security qualities. To our knowledge little attention has been paid to such research issue, although extensive works have been achieved on developing measurement properties for (traditional) internal software attributes [4], [5], [8], [11], [21], [23]. Some of the few works on this issue include [22] and [12]. In [22], Wang and Wulf motivate the rationale for theoretical validation of security metrics and highlight possible security attributes. However measurement concepts are barely defined or formalized. In [12], Millen proposed a theoretical framework for the definition and validation of survivability measures based on service-oriented architecture; note that survivability is an external software attribute. In this work, based on popular security design principles, we identify a number of internal software attributes, which relate to security facets as external software quality attributes. For each internal attribute, we propose a number of measurement properties that can be used to derive or validate theoretically related metrics. We show the feasibility of our framework by conducting case studies based on the attack surface metrics system [6], [10] and the privilege graph paradigm [2], which have been studied empirically in previous research. The remainder of the paper is structured as follows. In Section II, we present and discuss previous research directly influencing this work. In Section III, we present security design principles and software security attributes. We also discuss how the security design principles can be used to identify security-related internal attributes. In Section IV, we present a generic software model that we use as a formal abstraction to express security measurement properties for internal software attributes. In Section V, we present our case studies, by deriving sample metrics and discussing corresponding theoretical validation results. Finally, in Section VI, we make concluding remarks.

2 Background

In the last two decades, several efforts have been made towards rigorous definition of software attributes and their measures. While some of these works emphasize the application of traditional measurement theory to software metrics [5], [8], [11], others focus on formally defining expected properties of software attributes within an axiomatic framework [1], [21], [23]. For instance, Kitchenham et al. proposed a validation framework for software measure based on measurement theory, and centered on the notion of software entities and software attributes [8]. They defined several criteria for theoretical and empirical validation of software measures. 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" [8]. As this came as a shortcoming of previous axiomatic approaches, it triggered a discussion between the authors and some of the tenants of the axiomatic approaches [9], [14]. 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 measures will simply lead to abstracting away important relevant information, weakening as a consequence the checking capabilities of corresponding validation framework [14]. 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" [14]. According to them, deriving and associating properties with different measurement scales can define an attribute. In this case, given a software attribute measure, 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 measures of software attributes at different scales [13]. 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. In this paper, we build on previous works to define an axiomatic framework for theoretical validation of internal software security measures.

3 Software Security Concepts

3.1 Security design principles

A great deal of wisdom regarding secure system development has been gathered in the past decades. This has led to the definition of several security design principles, which are currently used for the design and implementation of secure systems [16]. Table 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

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.
Secure Failure	When a system fails its behavior becomes more insecure. So it is important to ensure that the system fails securely.

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 fulfil 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.

3.2 Software security attributes

3.2.1 Overview

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. 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, specifiers, testers), who have access to software work products such as code, specification, or documentation. Examples of internal software attributes include readability, complexity, modularity and so on. 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. 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 good basis for the development of an initial software security metrics program.

3.2.2 External software security attributes

Software security is a complex and multifaceted notion, 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 [15]. 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 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 measures 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 measures 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 under attack or failure its intended mission, in a timely fashion [12]. Attackability is a concept proposed by Howard and colleagues to measure the extent that a software system could be the target of successful attacks [6]. They define attackability as the short word for attack opportunity. More specifically, attackability is the likelihood that a successful attack will happen.

3.2.3 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 predict 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.
- *Excess Privilege:* captures the amount of excessive privileges that a software service may grant compared to its required privileges.
- *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 these 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 the focus on traditional qualities such as reliability and maintainability, not security. Excess privilege was introduced by T. Smith [18] 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 [3]. L. Smith reported on pilot activities to investigate the practicality of developing metrics for determining the strength of cryptographic algorithms [19]. 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 don't 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 one mentioned earlier. Mechanism strength for regular software services are computed by combining the strength of the individual mechanisms involved.

3.2.4 Guidelines 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. Prediction 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. The models are evaluated through empirical investigation. Although this is not the focus of this paper, we can draw from the previous analysis some guidelines that can be used to analyze empirically the relationship between the internal and external security measurement concepts. More specifically, we define four guidelines based on intuitive understanding of the security design principles as follows:

- ***Guideline 1:*** Security decreases as Service Complexity increases.
- ***Guideline 2:*** Security decreases as Service Coupling increases.

- ***Guideline 3:*** Security decreases as Excess Privilege increases.
- ***Guideline 4:*** Security increases as Mechanism Strength increases

These guidelines may serve as hypothesis in developing the prediction models. The dependent and independent variables may correspond to external and internal attributes respectively.

4 Security Measurement Properties

We identify and define, in this section, some useful properties for the internal attributes proposed in the previous section. We use an abstract software model based on service-oriented architecture to express these properties. First, we introduce our software model and then present the measurement properties.

4.1 Software model

Typically, a software system involves multiple services. Either legitimate or illegitimate users usually take advantages of deficiencies and vulnerabilities in 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 [12].

4.1.1 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. [1], who use the term elements instead of components. Unlike Briand et al., who use a modular software model, we use in this paper a service-oriented software model, which fits well with security analysis purposes. In [12], 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 $s \in S$, $\overline{s} \in \overline{S}$ and $\underline{s} \subseteq \overline{S}$ denote respectively the service containing s and the support service set for s. An atomic service is

a service, which has no supporting services. Formally, $\forall s \in S, \overline{s}$ is an atomic service if an only if $\underline{s} = \emptyset$. Services and configurations have the following properties:

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

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

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 $\overline{s}$ has only one configuration: $\overline{s} = \{s\}$. This is expressed formally by the following proposition.

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

Proof: Based on the earlier definition $\forall s \in S\ \overline{s}\ is\ atomic \Leftrightarrow \underline{s} = \emptyset$. So we simply need to show that $\forall s \in S\ \underline{s} = \emptyset \Rightarrow \overline{s} = \{s\}$. We can achieve that by contradiction. Let us assume that $\exists s \in S\ \underline{s} = \emptyset \wedge \overline{s} \neq \{s\}$; this will imply that $\exists s' \in S\ (s \neq s') \wedge (\{s, s'\} \subseteq \overline{s}) \wedge (\underline{s} = \emptyset) \Rightarrow \exists s' \in S\ (s \neq s') \wedge (\overline{s} = \overline{s'}) \wedge (\underline{s} \subseteq \underline{s'})$. That is in contradiction with irredundancy property (2)

The relationships between two configurations correspond to the relationships between their respective components. Given a service configuration s, let $\mu(s)$ 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 $\overline{s_1}$ and $\overline{s_2}$ as: $\overline{s_1} \propto \overline{s_2} = \bigcup_{t_1 \in \overline{s_1}, t_2 \in \overline{s_2}} (t_1 \propto t_2)$.

4.1.2 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 $<S, \overline{S}_{Requestor}, \Gamma, \wp, priv, pol>$. 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 $\overline{s} \in \overline{S}$ and a security mechanism $r \in \Gamma$, we use $r \triangleright \overline{s}$ to denote that the security mechanism r protects $\overline{s}$. 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 $\oplus : \Gamma \times \Gamma \rightarrow \Gamma$ and $\otimes : \Gamma \times \Gamma \rightarrow \Gamma$, we use $r_1 \otimes r_2 \triangleright \overline{s}$ to denote that r_1 and r_2 work in parallel to protect service $\overline{s}$ and $r_1 \oplus r_2 \triangleright \overline{s}$ to denote that r_1 and r_2 work in sequence to protect service $\overline{s}$. 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 $\overline{S}_{\mathrm{Requestor}}$ 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 are 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 [17]. 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 $priv : \overline{S}_{\mathrm{Requestor}} \times \overline{S} \rightarrow 2^{\wp}$, 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: $pol : \overline{S}_{\mathrm{Requestor}} \times \overline{S} \rightarrow 2^{\wp}$. 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.

4.1.3 Example

As an example, Figure 4.1 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; it is up to the designer to decide the level of granularity of service specification [12].

In Figure 4.1, 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

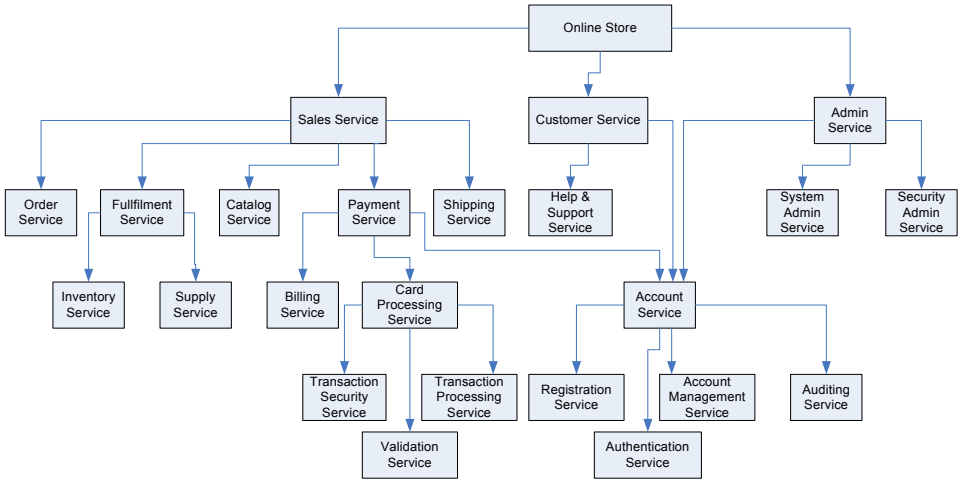

Figure 4.1: Service-oriented architecture for an Online Retail Store. An arrow pointing from service X to service Y indicates that Y is a supporting service for one of the configurations of X.

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.

4.2 Measurements 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. Establishing completeness is a difficult

task, which is beyond the scope of this work.

4.2.1 Service complexity

In [11], Melton et al. discuss the 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 paper only to structural complexity; the reader is referred to [11] for detail 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, $\wp$*, priv, pol*>, the *complexity* of a software service $\overline{s} \in \overline{S}$ is defined by a function denoted by *Complexity(*$\overline{s}$*).* 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, $\wp$*, priv, pol*>, we expect these functions to satisfy the following properties:

1. Axiom AC1: $\forall \overline{s} \in \overline{S}\ Complexity(\overline{s}) \geq 0$
2. Axiom AC2: $\forall \overline{s} \in \overline{S}\ \forall t \in \overline{s}\ Complexity(\overline{s}) \geq Complexity(t)$
3. Axiom AC3: $\forall s \in S\ \forall \overline{t} \in \underline{s}\ Complexity(\overline{t}) \leq Complexity(s)$
4. Axiom AC4: $\forall s \in S\ \underline{s} = \emptyset \Rightarrow Complexity(\overline{s}) = Complexity(s)$
5. Axiom AC5: $\forall \overline{s}, \overline{t} \in \overline{S}\ \overline{s} \subseteq \overline{t} \Rightarrow Complexity(\overline{s}) \leq Complexity(\overline{t})$

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:

Theorem TC1: $\forall s \in SComplexity(s) \geq 0$

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

Due to space limitation, we skip the proofs of theorems TC1 and TC2, as well as for theorems TD1-TD2, TP1-TP2, and TS1 which actually are straightforward.

4.2.2 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, $\overline{S}_{Requestor}$, Γ, $\wp$, priv, pol*>, the *coupling* between two software services $\overline{s_1}, \overline{s_2} \in \overline{S}$ is defined as a function denoted by *Coupling($\overline{s_1}, \overline{s_2}$)* 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: $\forall \overline{s_1}, \overline{s_2} \in \overline{S}\ \overline{s_1} \propto \overline{s_2} = \emptyset \Rightarrow Coupling(\overline{s_1}, \overline{s_2}) = 0$
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 null 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 null if all the couplings between their pairs of supporting services are null. These properties are expressed by theorems TD1 and TD2 defined as follows:

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

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

4.2.3 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 he needs in order to accomplish his 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 and over simplification. The concept of *Excess Privilege* originally introduced by Terry A. Smith [18] 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,* $\overline{S}_{\mathrm{Requestor}}$, 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: $\forall \overline{s} \in \overline{S} \; EP(\overline{s}) \geq 0$
2. Axiom AP2: $\forall s \in S \; \forall \overline{t} \in \underline{s} \; EP(s) \geq EP(\overline{t})$
3. Axiom AP3: $\forall s \in S \; \underline{s} \neq \emptyset \; \Rightarrow EP(s) \leq \sum_{\overline{t_i} \in \underline{s}} EP(\overline{t_i})$
4. Axiom AP4: $\forall s \in S \; EP(\overline{s}) \geq EP(s)$
5. Axiom AP5: $\forall s \in S \; \underline{s} = \emptyset \Rightarrow EP(\overline{s}) = EP(s)$

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.

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

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

4.2.4 Mechanisms 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,* $\overline{S}_{\mathrm{Requestor}}$, Γ, $\wp$*, priv, pol*>, we define the *Mechanism Strength* of a software service $\overline{s} \in \overline{S}$, as a function *MStrength(*$\overline{s}$*)*, 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: $\forall \overline{s} \in \overline{S} \; MStrength(\overline{s}) \geq 0$
2. Axiom AS2: $\forall s \in S \; \underline{s} = \emptyset \Rightarrow MStrength(\overline{s}) = MStrength(s)$
3. Axiom AS3: $\forall s \in S \; [\overline{s} \in (\overline{S} - \Gamma) \wedge \underline{s} = \emptyset] \; \Rightarrow MStrength(\overline{s}) = 0$
4. Axiom AS4: $\forall s \in S \; \underline{s} \neq \emptyset$
   $\Rightarrow [(\forall \overline{t} \in \underline{s} \; MStrength(\overline{t}) = 0) \Rightarrow MStrength(s) = 0]$
5. Axiom AS5: $\forall \overline{s} \in \overline{S} \; (\forall t \in \overline{s} \; MStrength(t) = 0) \Rightarrow MStrength(\overline{s}) = 0$
6. Axiom AS6: $\forall \overline{s} \in \overline{S}, m \subset \Gamma \; \otimes m \triangleright \overline{s} \Rightarrow MStrength(\overline{s}) \leq \underset{r \in m}{Min}\{MStrength(r)\}$
7. Axiom AS7: $\forall \overline{s} \in \overline{S}, m \subset \Gamma \; \oplus m \triangleright \overline{s} \Rightarrow MStrength(\overline{s}) \geq \underset{r \in m}{Max}\{MStrength(r)\}$

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 null (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 null (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 null (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 null. This is expressed by theorem TS1:

Theorem TS1: $\Gamma = \emptyset \Rightarrow \forall \overline{s} \in \overline{S}\ MStrength(\overline{s}) = 0$

5 Case Studies

In this section, we illustrate the properties introduced above by presenting case studies based on existing measurement frameworks. The first case study, based on the attack surface metrics proposed in [6, 10], allows the derivation and evaluation of sample coupling, complexity and excess privileges metrics. The second study, based on the privilege graph paradigm [2], allows the derivation and evaluation of sample mechanism strength metrics.

5.1 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 [10], we derive a collection of metrics that match three of our internal attributes, namely service coupling, service complexity, and excess privilege.

5.1.1 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 [6], [10], 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*$\times$ *Resource*$\times$ *Rights* based on Lampson's access matrix model. 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,* $\overline{S}_{\mathrm{Requestor}}$, Γ, $\wp$*, 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. *(* 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 *(* can be derived from the access matrixes of the attack surface system.

5.1.2 Attack surface metrics

In [10], the attack surface is defined as a tuple $\left(A_S, \bigcup_{a \in A_S} \mathrm{Res}(a)\right)$, 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*$\rightarrow$ *[0,1]*. Formally, the attack surface is defined as:

$$attack\_surf = \sum_{i=1}^{n} count(S_i) \times payoff(S_i) \tag{5.1}$$

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 [10], it is suggested that not all attack classes need to be involved in the measure of an attack surface.

5.1.3 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 metrics 5.1 to software services and configurations. Specifically, given a set of top attack classes S_{Attack}, the attack surface of a software service $\overline{s}$ (resp. a configuration s) can be defined as follows:

$$attack\_surf_1(\overline{s}) = \sum_{a \in S_{Attack}} count(a \bullet \overline{s}) \times payoff(a)$$
$$attack\_surf_1(s) = \begin{cases} \sum_{a \in S_{Attack}} count(\bigcup_{\overline{t} \in \underline{s}} (a \bullet \overline{t})) \times payoff(a) (\text{if} \underline{s} \neq \emptyset) \\ attack\_surf_1(\overline{s}) (\text{if} \underline{s} = \emptyset) \end{cases} \quad (5.2)$$

Where $a \bullet \overline{s}$ (resp. $a \bullet s$) denotes the set of attack instances of type a related to service $\overline{s}$ (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 regular 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 minimum. 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 $\overline{s_t} \in \overline{S}$ (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:

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

$$attack\_surface_2(s_t, s_u)$$

$$= \begin{cases} cardinality(\underset{\overline{t_i} \in \underline{s_t}}{\cup} resource(\overline{t_i}) \cap \underset{\overline{t_j} \in \underline{s_u}}{\cup} resource(\overline{t_j}))(\text{if} \underline{s_t} \neq \emptyset \wedge \underline{s_u} \neq \emptyset) \\ cardinality(\underset{\overline{t_i} \in \underline{s_t}}{\cup} resource(\overline{t_i}) \cap resource(\overline{s_u}))(\text{if} \underline{s_t} \neq \emptyset \wedge \underline{s_u} = \emptyset) \\ cardinality(resource(\overline{s_t}) \cap \underset{\overline{t_j} \in \underline{s_u}}{\cup} resource(\overline{t_j}))(\text{if} \underline{s_t} = \emptyset \wedge \underline{s_u} \neq \emptyset) \\ attack\_surface_2(\overline{s_t}, \overline{s_u})(\text{if} \underline{s_t} = \emptyset \wedge \underline{s_u} = \emptyset) \end{cases} \tag{5.3}$$

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 measure 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 a delivery of a software service to a given requestor: *actual privileges, allowed privileges*, and *least privileges* [18]. 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 [16], [18]. 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 [10], 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 [10], 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 $\overline{s}$, we can derive an attack surface metric as follows:

$$attack\_surface_3(\overline{s}) =$$
$$cardinality(\bigcup_{u \in principle(\overline{s})} (\bigcup_{res_i \in resource(\overline{s})} u \times res_i \times right(u, res_i) - LP(u, \overline{s})))$$
$$attack\_surface_3(s) =$$
$$\begin{cases} attack\_surface_3(\overline{s}) \ (if \ \underline{s} = \emptyset) \\ cardinality(\bigcup_{\overline{t} \in \underline{s}} \bigcup_{u \in principle(\overline{t})} (\bigcup_{res_i \in resource(\overline{t})} u \times res_i \times right(u, res_i) - LP(u, \overline{t}))) \\ (if \ \underline{s} \neq \emptyset) \end{cases} \tag{5.4}$$

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

5.2 Privilege graph paradigm

5.2.1 Foundation

It has been established by Dacier and Deswartes that the vulnerabilities of a computing system can be described using a privilege graph [2]. 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 [2], 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.

5.2.2 The MTTF metrics

In [2], 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 [2], 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:

$$\begin{aligned} MTTF(i) &= t_i + \sum_{k \in out(i)} P_{ik} \times MTTF(k) \\ t_i &= \sum_{k \in out(i)} 1/\lambda_{ik} \qquad P_{ik} = \lambda_{ik} \times t_i \end{aligned} \tag{5.5}$$

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.

5.2.3 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 measure 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 $\overline{s} \in \overline{S}$ and the set of intrusion process state graphs $Q_{\overline{s}}$ whose targets represent the service$\overline{s}$, we define a *Minimum MTTF metric* (MinMTTF) for service $\overline{s}$ (resp. configuration s) as follows:

$$\begin{aligned} MinMTTF(\overline{s}) &= \underset{q \in Q_{\overline{s}}}{Min}\{MTTF(I_q)\} \\ MinMTTF(s) &= \begin{cases} \sum\limits_{\overline{t_i} \in \underline{s}} MinMTTF(\overline{t_i}) (\text{if} \underline{s} \neq \emptyset) \\ MinMTTF(\overline{s}) (\text{if} \underline{s} = \emptyset) \end{cases} \end{aligned} \tag{5.6}$$

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

5.3 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 definitional flaws, 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. 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.

5.3.1 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:

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

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:

$$\begin{aligned}
&Postulate\ P11 \\
&\Rightarrow \forall s \in S \; \left[\bigcup_{\overline{t} \in \underline{s}} (a \bullet \overline{t}) \right] \subseteq (a \bullet \overline{s}) \\
&\Rightarrow \forall a \in S_{Attack}\ count(\left[\bigcup_{\overline{t} \in \underline{s}} (a \bullet \overline{t}) \right]) \leq count(a \bullet \overline{s}) \\
&\Rightarrow \forall a \in S_{Attack}\ count(\left[\bigcup_{\overline{t} \in \underline{s}} (a \bullet \overline{t}) \right]) \times payoff(a) \leq count(a \bullet \overline{s}) \times payoff(a) \\
&\Rightarrow \sum_{a \in S_{attack}} count(\left[\bigcup_{\overline{t} \in \underline{s}} (a \bullet \overline{t}) \right]) \times payoff(a) \leq \sum_{a \in S_{attack}} count(a \bullet \overline{s}) \times payoff(a) \\
&\Rightarrow \forall \overline{s} \in \overline{S} \; \forall t \in \overline{s} \; attack_surf_1(t) \leq attack_surf_1(\overline{s}).
\end{aligned} \tag{5.7}$$

5.3.2 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:

$$PostulateP21 : \forall \overline{s} \in \overline{S}\ resource(\overline{s}) = \bigcup_{t \in \overline{s}} resource(t)$$

$$PostulateP22 : \forall s \in S\ resource(s) = \bigcup_{\overline{t} \in \underline{s}} resource(\overline{t})$$

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:

$$\begin{aligned}
&1)\ Postulate\ P22\\
&\Rightarrow \forall s_1, s_2 \in S\ attack_surface_2(s_1, s_2) = resource(s_1) \cap resource(s_2)\\
&2)\ Postulate\ P22\\
&\Rightarrow \forall s_1, s_2 \in S\ resource(s_1) \cap resource(s_2)\\
&\qquad = \left[\bigcup_{\overline{s}_i \in \underline{s_1}} resource(\overline{s}_i)\right] \cap \left[\bigcup_{\overline{s}_j \in \underline{s_2}} resource(\overline{s}_j)\right]\\
&\Rightarrow resource(s_1) \cap resource(s_2) = \bigcup_{\overline{s}_i \in \underline{s_1}, \overline{s}_j \in \underline{s_2}} [resource(\overline{s_i}) \cap resource(\overline{s_j})]\\
&\Rightarrow cardinality(resource(s_1) \cap resource(s_2))\\
&\qquad = cardinality\left(\bigcup_{\overline{s}_i \in \underline{s_1}, \overline{s}_j \in \underline{s_2}} [resource(\overline{s_i}) \cap resource(\overline{s_j})]\right)\\
&\Rightarrow cardinality(resource(s_1) \cap resource(s_2))\\
&\qquad \leq \sum_{\overline{s}_i \in \underline{s_1}} \sum_{\overline{s}_j \in \underline{s_2}} cardinality(resource(\overline{s}_i) \cap resource(\overline{s}_j))\\
&1)\ and\ 2) \Rightarrow \forall s_1, s_2 \in S\ attack_surface_2(s_1, s_2)\\
&\qquad \leq \sum_{\overline{s}_i \in \underline{s_1}} \sum_{\overline{s}_j \in \underline{s_2}} attack_surface_2(\overline{s_i}, \overline{s_j}).
\end{aligned}$$

5.3.3 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:

$$PostulateP31 : \forall s \in S\ \forall \overline{t} \in \underline{s}\ \forall u \in \Pr inciple(\overline{t}),$$
$$(\bigcup_{res_i \in Resource(\overline{t})} u \times res_i \times right(u, res_i) - LP(u, \overline{t}))$$
$$\subseteq (\bigcup_{res_i \in Resource(\overline{s})} u \times res_i \times right(u, res_i) - LP(u, \overline{s}))$$

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:

$$1)\ \forall s \in S\ \underline{s} \neq \emptyset$$
$$\Rightarrow \bigcup_{\overline{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\overline{t})} (\bigcup_{res_i \in Resource(\overline{t})} u \times res_i \times right(u, res_i) - LP(u, \overline{t}))$$
$$\subseteq \bigcup_{u \in \Pr inciple(\overline{s})} (\bigcup_{res_i \in Resource(\overline{s})} u \times res_i \times right(u, res_i) - LP(u, \overline{s}))\ (P31)$$
$$\Rightarrow$$
$$cardinality(\bigcup_{\overline{t} \in \underline{s}} \bigcup_{u \in \Pr inciple(\overline{t})} \{\bigcup_{res_i \in Resource(\overline{t})} u \times res_i \times right(u, res_i) - LP(u, \overline{t})\})$$
$$\leq cardinality(\bigcup_{u \in \Pr inciple(\overline{s})} \{\bigcup_{res_i \in Resource(\overline{s})} u \times res_i \times right(u, res_i) - LP(u, \overline{s})\})$$
$$\Rightarrow attack\_surface_3(s) \leq attack\_surface_3(\overline{s})$$
$$2)\ \forall s \in S\ \underline{s} = \emptyset \Rightarrow attack\_surface_3(s) = attack\_surface_3(\overline{s})\ \ (by\ definition)$$
$$1)\ and\ 2) \Rightarrow \forall s \in S\ attack\_surface_3(s) \leq attack\_surface_3(\overline{s}).$$

5.3.4 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:

$$PostulateP41 : \forall \overline{s} \in \overline{S}\ \forall t \in \overline{s}\ MinMTTF(\overline{s}) = \underset{t \in \overline{s}}{Min}\{MinMTTF(t)\}$$
$$LemmaL41 : \forall \overline{s} \in \overline{S} \forall q \in Q_{\overline{s}}, MTTF(I_q) \leq \underset{i \in path(q)}{Min}\{MTTF_i(I_q)\}$$
$$LemmaL42 : \forall \overline{s} \in \overline{S} \forall q \in Q_{\overline{s}} \forall i_1, i_2 \in path(q)\ i_1 \subseteq i_2$$
$$\Rightarrow MTTF_{i_1}(I_q) \leq MTTF_{i_2}(I_q)$$

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 [2] 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:

$Let\ i \bullet p\ denote\ a\ subpath\ of\ pathp\ where\ i \bullet p\ ends\ at\ the\ node\ i\ of\ p.$
$1) \forall \overline{s} \in \overline{S}, m \subset \Gamma\ \otimes m \triangleright \overline{s}$
$\Rightarrow (\forall q \in Q_{\overline{s}}\ \forall r_i, r_j \in m\ r_i \neq r_j \Rightarrow r_i, r_j\ protect\ \overline{s}\ in\ different\ paths\ of\ q)$
$\Rightarrow \forall q \in Q_{\overline{s}} \forall i \in path(q) \forall r \in m,$
$if\ r\ is\ contained\ in\ path\ i,\ MTTF_{r \bullet i}(I_q) = MTTF_i(I_q).$
$2) LemmaL41$
$\Rightarrow MinMTTF(\overline{s}) = \underset{q \in Q_{\overline{s}}}{Min}\{MTTF(I_q)\} \leq \underset{q \in Q_{\overline{s}}}{Min}\{\underset{i \in path(q)}{Min}\{MTTF_i(I_q)\}\}$
$Based\ on\ 1),\ 2),\ we\ can\ deduct\ the\ following:$
$\forall \overline{s} \in \overline{S}, m \subset \Gamma\ \otimes m \triangleright \overline{s}$
$\Rightarrow MinMTTF(\overline{s}) \leq \underset{q \in Q_{\overline{s}}}{Min}\{\underset{i \in path(q)}{Min}\{MTTF_{r \bullet i}(I_q)\}\}$
$= \underset{r \in m}{Min}\{\underset{q \in Q_{\overline{s}}}{Min}\{MTTF_{r \bullet i}(I_q)\}\}$
$= \underset{r \in m}{Min}\{MinMTTF(r)\}.$

5.4 Summary

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.

6 Conclusion

Theoretical validation of software security measures is an important step towards their general acceptance. We define and formalize in this paper a collection of properties characterizing security-related internal software attributes. The 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. The framework provides a rigorous ground for identifying and evaluating new security metrics. Like in most previous works on measurement properties, our proposed formal framework is sound but not complete. Due to the complexity of software security, completeness is difficult to establish. Our properties should be interpreted as necessary but not sufficient. It is important to note that the internal attributes suggested in this paper 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.

Acknowledgments

This work was partly supported by the Natural Science and Engineering Research Council of Canada (NSERC) through a Discovery grant. We would like to thank Dr. J. H. Weber-Jahnke from the University of Victoria, Dr. I. Ryl from the University of Lille (France), and Mr. A. M. Hoole from the University of Victoria for their suggestions and recommendations on the preliminary version of this paper.

References

[1] L. C. Briand, S. Morasca, V. R. Basili, Property-based software engineering measurement, *IEEE TSE*, **22**(1996), 68-86.

[2] M. Dacier, Y. Deswarte, Privilege graph: an extension to the typed access matrix model, *Lecture Notes in Computer Science*, **875**(1994), 319–334.

[3] J. H. P. Eloff, Selection process for security packages, *Computers & Security*, **2**(1983), 159-167.

[4] N. Fenton, *Software Metrics: A rigorous Approach*, Chapman & Hall, 1991.

[5] N. Fenton, Software measurement: a necessary scientific basis, *IEEE TSE*, **20**(1994), 199-206.

[6] M. Howard, J. Pincus, J. M. Wing, Measuring relative attack surfaces, *Proceeding of Workshop on Advanced Developments in Software and System Security*, (2003), 109-137.

[7] D. Frank, Agencies Seek Security Measures, *CIO Magazine: http://www.cio.com*, 2000.

[8] B. Kitchenham, S. L. Pfleeger, N. Fenton, Towards a framework for software measurement validation, *IEEE TSE*, **21**(1995), 929-943.

[9] B. Kitchenham, S. L. Pfleege, N. Fenton, Reply to: Comments on towards a framework for software measurement validation, *IEEE TSE*, **23**(1997), 187-189.

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

[11] A. C. Melton, D. A. Gustafson, J. M. Bieman, A. L. Baker, A mathematical perspective for software measures research, *Software Eng. J.*, **5**(1990), 246-254.

[12] J. K. Millen. Survivability Measure, *Research Report, Computer Science Laboratory (CSL)*, SRI, CA, USA.

[13] S. Morasca, L. C. Briand, Towards a theoretical framework for measuring software attributes, *Fourth International Software Metrics Symposium (METRICS'97)*, (1997), 68-86.

[14] 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*, **23**(1997), 187-188.

[15] D. M. Nicol, W. H. Sanders, K. S. Trivedi, Model-based evaluation: from dependability to security, *IEEE TDSC*, **1**(2004), 48-65.

[16] J. Saltzer, M. Schroeder, The protection of information in computer systems, *Proc. of the IEEE*, **63**(1975), 1278-1308.

[17] R. S. Sandhu, The typed access matrix model, *Proc 1992 IEEE Symposium on research in Security and Privacy*, May 4-6, (1992), 122-136.

[18] T. A. Smith, User definable domains as a mechanism for implementing the least privilege principle, *Proceedings of 9th National Computer Security Conference*, (1986), 143-148.

[19] L. Smith, Cryptographic algorithm metrics, *NIST and CSSPAB Workshop*, (2000), 13-14.

[20] J. Tian, M. V. Zelkowitz, A formal program complexity model and its applications, *J. Syst. Soft.*, **17**(1992), 253-266.

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

[22] C. Wang, W. A. Wulf, Towards a framework of security measuremen, *20th NISSC Proceedings*, Baltimore, Maryland, (1997), 522-533.

[23] E. J. Weyuker, Evaluating software complexity measures, *IEEE TSE*, **14**(1988), 1357-1365.